An Investigation of the Factors Affecting the Lifecycle Costs of COTS-Based Systems

By

Laurence Michael Dunn

University of Portsmouth

School of Computing

This thesis is been submitted in partial fulfilment of the requirements for the award of the degree of Doctor of Philosophy of the University of Portsmouth.

June 2011
Abstract

This research used a case study based approach to test Abts’ *CBS Functional Density Rule of Thumb* (Abts, 2002; 2004) and to identify the factors influencing the costs of COTS-based systems by means of statistical analysis of a large component dataset from IBM, Grounded Theory analysis of a series of interviews with software architects and project managers and an extended literature review.

Whilst the use of Glaser’s (1978) Grounded Theory approach provided support for Abts’ theory the statistical analysis provided no support for the theory that maximising the amount of system functionality provided by COTS components reduced system development costs.

This has led to the identification of a weakness in the Grounded Theory method in that it is unable to move beyond the preconceptions of the interviewees if the interview data collection method is used in isolation of other data collection methods.

However, overall this research has provided a deeper understanding of the issues affecting COTS-based design. By combining the outcomes of the Grounded Theory analysis and literature review a series of forces influencing the costs of building COTS-based systems have been identified, together with a set of principles, which when used in combination can enable software practitioners to make informed decisions about the impact on costs of using components.

Keywords

COTS; Commercial-off-the-shelf; Grounded Theory; Culture, Components; CBS Functional Density Rule of Thumb.
Acknowledgements

I would like to thank the following people for their help, without which this thesis would not have been completed.

Dr Tineke Fitch and Jane Chandler for their invaluable help in getting this project completed;
Bernie Higgins and Graham Elliott for their help with the statistical analysis;
Hugh Varily, IBM Distinguished Engineer, for supporting this project;
Dr George Allan for his supervisory assistance and support in starting this project;
and finally, for the continued support of my partner, Soraya.
Table of Contents

Abstract ............................................................................................................................. i
Keywords ........................................................................................................................ i
Acknowledgements ......................................................................................................... ii
Table of Contents ........................................................................................................... iii
List of Tables .................................................................................................................. xi
List of Figures ................................................................................................................ xiii
Declaration ..................................................................................................................... xvi
Chapter 1 Setting the Scene ......................................................................................... 1
  1.1 Introduction .............................................................................................................. 1
  1.2 CBS Functional Density Rule of Thumb ............................................................. 3
  1.3 Aim of the Research ............................................................................................ 4
  1.4 Research Objectives ........................................................................................... 4
  1.5 Research Questions ............................................................................................ 4
  1.6 Research Plan ....................................................................................................... 5
  1.7 Contribution to Knowledge ................................................................................. 6
  1.8 Definition of Terms ........................................................................................... 7
  1.9 Conclusions .......................................................................................................... 7
Chapter 2 Literature Review ......................................................................................... 8
  2.1 Introduction .......................................................................................................... 8
  2.2 Definition of a COTS Software Component ...................................................... 8
    2.2.1 Software Components .................................................................................. 8
    2.2.2 Interface ....................................................................................................... 11
    2.2.3 Code Reuse .............................................................................................. 11
    2.2.4 COTS Software Components ..................................................................... 11
    2.2.5 Black Box Nature of COTS Software Components ................................... 13
    2.2.6 Glue Code and Wrapper Code .................................................................... 15
    2.2.7 COTS Component Definition used in this Thesis ....................................... 16
  2.3 Conceptual Justification of COTS Software Component Use .......................... 16
  2.4 Historical Context of using COTS Software Components .............................. 18
  2.5 Perceived Benefits of COTS-Based Software Engineering ............................. 20
  2.6 Issues with COTS-Based Development ............................................................. 22
    2.6.1 Development Approach ............................................................................ 22
2.6.2 COTS Component Selection and Acquisition ............................................. 24
2.6.3 Coupling and Cohesion ........................................................................... 25
2.6.4 Component Selection ............................................................................. 26

2.7 Maintenance of COTS-Based Systems ...................................................... 29
  2.7.1 Factors Influencing the Cost of Maintaining COTS-Based Systems ....... 31
  2.7.2 Strategies for Reducing Maintenance Costs ....................................... 32

2.8 Challenges to the Perceived Benefits of CBD ......................................... 34

2.9 Conclusions ............................................................................................... 35

Chapter 3 Research Methods ........................................................................... 36
  3.1 Introduction ............................................................................................... 36
  3.2 Adoption of a Case Study Approach ....................................................... 36
  3.3 Justification for Using the Case Study Research Method ....................... 38

3.4 Data Collection ........................................................................................... 41
  3.4.1 Triangulation ....................................................................................... 41
  3.4.2 Identifying Data Sources ..................................................................... 42

3.5 Statistical Analysis ..................................................................................... 43
  3.5.1 Requested Data Types ....................................................................... 43
  3.5.2 Data Analysis Design ......................................................................... 45
  3.5.3 Hypotheses Tested .............................................................................. 46
  3.5.4 Inferential Statistical Tests ................................................................... 46

3.6 Grounded Theory ....................................................................................... 47
  3.6.1 Data Collection Methods .................................................................... 48
    3.6.1.1 Interview Data Collection Method .................................................. 48
    3.6.1.2 Direct Observation ....................................................................... 49
    3.6.1.3 Document Search ......................................................................... 49
  3.6.2 Data analysis Approach ....................................................................... 49

3.7 Conclusions ................................................................................................ 58

Chapter 4 Statistical Analysis ......................................................................... 59
  4.1 Introduction ............................................................................................... 59
Memo on Concept Increasing cost................................................................. 403
Memo on Concept Increasing effort .............................................................. 406
Memo on Concept Knock-on-effect ............................................................. 410
Memo on Concept Losing faith .................................................................. 411
Memo on Concept Maintenance complexity .............................................. 412
Memo on Concept Managing change ......................................................... 416
Memo on Concept Organisational concerns ............................................. 417
Memo on Concept Redoing integration work ............................................. 418
Memo on Concept Reducing integration effort .......................................... 420
Memo on Concept Reducing potential problems ...................................... 422
Memo on Concept Reducing user intervention ........................................... 423
Memo on Concepts Relationship complexity ... ....................................... 424
Memo on Concept Resisting change ......................................................... 428
Memo on Concept Support quality ............................................................ 429
Memo on Concept System complexity ....................................................... 431
Memo on Concept Vendor homogeneity .................................................... 434
List of Tables

Table 3.1: Glaser’s eighteen theoretical coding families (Glaser, 1978, pp. 72-82)..... 57

Table 4.1: Summary statistics for n = 158 systems, where Log(xx) = measured on a logarithmic scale to base 10................................................................. 68
Table 4.2: Table of Coefficients for dependent variable Log(Development effort)..... 79
Table 4.3: Table of Coefficients for dependent variable Log(Maintenance effort).... 80
Table 4.4: Table of Coefficients for dependent variable Log(Total effort)............. 81

Table 5.1: Key text, key text IDs and open codes from Interview A .................... 100
Table 5.2: Emergence of concepts from open codes in Interview A................. 105
Table 5.3: Emergence of categories from Interview A................................... 107
Table 5.4: List of concepts which emerged from all interviews, including key text identifiers................................................................. 110
Table 5.5: List of categories and associated concepts, which emerged from all of the interviews. .................................................................................. 111

Table A1: Table presenting the raw data, sorted on variable Percentage of COTS components.................................................................................. 245

Table A2a: Table A2a. System size categories and their equivalent Function Point values (Software Measurement Services Ltd, 2005).............................. 246
Table A2b: IBM System size categories and their equivalent FP size (Peter Thomas, IFPUG Certified Function Point Specialist, personal communication, April 01, 2008). .................................................................................. 246

Table A4a: Descriptive statistics presenting the range, minimum, mean, median, mode, standard deviation, variance, skewness and kurtosis of the raw data for variables:
Table A4b: Results of Kolmogorov-Smirnov test of normality for variables: Development effort, Percentage of COTS components, System size, Maintenance effort and Total effort. .......................................................... ................................................................................. 251

Table A7a: ANOVA table assessing the significance of the multiple linear regression model to determine if Log(System size) and Percentage of COTS components are significant predictors of Log(Development effort). .......................................................... 270

Table A7b: Model summary table showing effect of predictor variables Log(System size) and Percentage of COTS components on dependent variable Log(Development effort). ................................................................................. 270

Table A7c: Table of Coefficients for dependent variable Log(Development effort). ................................................................................. 271

Table A7d: ANOVA table assessing the significance of the multiple linear regression model to determine if Log(System size) and Percentage of COTS components are significant predictors of Log(Maintenance effort). .......................................................... 271

Table A7e: Model summary table showing effect of predictor variables Log(System size) and Percentage of COTS components on dependent variable Log(Maintenance effort). ................................................................................. 272

Table A7f: Table of Coefficients for dependent variable Log(Maintenance effort). ................................................................................. 272

Table A7g: ANOVA table assessing the significance of the multiple linear regression model to determine if Log(System size) and Percentage of COTS components are significant predictors of Log(Total effort). ................................................................................. 272

Table A7h: Model summary table showing effect of predictor variables Log(System size) and Percentage of COTS components on dependent variable Log(Total effort). ................................................................................. 273

Table A7i: Table of Coefficients for dependent variable Log(Total effort). ................................................................................. 273
List of Figures

Figure 1.1: Diagrammatic structure of the PhD research programme. .......................... 6
Figure 2.1: A black box approach to COTS components (Allan, 2003) ......................... 14

Figure 4.1: Scatter diagram plotting \textit{Percentage of COTS components} with \textit{Log(Development effort)}. ......................................................................................................................... 70
Figure 4.2: Scatter diagram plotting \textit{variables Percentage of COTS components} and \textit{Log(Maintenance effort)}. ....................................................................................................................... 71
Figure 4.3: Scatter diagram plotting \textit{Log(System size)} with \textit{Log(Development effort)}. 73
Figure 4.4: Scatter diagram plotting \textit{Log(System size)} with \textit{Log(Maintenance effort)}. 74
Figure 4.5: Scatter diagram plotting \textit{Percentage of COTS components} with \textit{Log(Total effort)}. ......................................................................................................................... 75
Figure 4.6: Scatter diagram plotting \textit{Log(System size)} with \textit{Log(Total effort)}. ........... 76

Figure 5.1: Overview of the interview data analysis process ........................................ 97
Figure 5.2: Cluster diagram providing diagrammatical representation of concepts and categories emerging from Interview A. ................................................................. 107
Figure 5.3: Cluster diagram providing a visual representation of the relationships between concepts and categories which emerged from the eleven interviews .......... 113
Figure 5.4: Diagrammatical representation of the relationship between concepts linking to category CONTROLLING COST ................................................................. 115
Figure 5.5: Diagrammatical representation of the relationship between concepts linking to category ORGANISATIONAL ISSUES .................................................. 130
Figure 5.6: Diagrammatical representation of the relationship between concepts linking to category DESIGN PRINCIPLES ......................................................... 137
Figure 5.7: Diagrammatical representation of the relationship between concepts linking to category MANAGING COMPLEXITY ......................................................... 143
Figure 5.8: Diagrammatical representation of the relationship between concepts linking to category MANAGING CHANGE ............................................................ 148
Figure 5.9: Diagrammatical representation of the relationship between concepts linking to category CULTURAL ISSUES ................................................................. 152
Figure 5.10: A visual representation of the relationship between categories. ............. 155
Figure 8.11: Cluster diagram providing a visual representation of the relationships between concepts and categories

Figure A3a: Bar diagram showing the values of Development effort for the systems sorted (from left to right) by Percentage of COTS components

Figure A3b: Bar diagram showing the values of Maintenance effort for the systems sorted (from left to right) by Percentage of COTS components

Figure A3c: Stacked bar diagram showing the values of Total effort (Development effort plus Maintenance effort) for the systems sorted (from left to right) by Percentage of COTS

Figure A4a: Normal Q-Q plot of variable Development effort

Figure A4b: Histogram showing the distribution of variable Development effort

Figure A4c: Normal Q-Q plot of variable Percentage of COTS components

Figure A4d: A histogram showing the distribution of variable Percentage of COTS components

Figure A4e: Normal Q-Q plot of variable System size

Figure A4f: Histogram showing distribution of variable System size

Figure A4g: Normal Q-Q plot of variable Maintenance effort

Figure A4h: Histogram showing distribution of Maintenance effort

Figure A4i: Normal Q-Q plot of variable Total effort

Figure A4j: Histogram showing distribution of Total effort

Figure A5a: Normal Q-Q Plot of Log(Development effort)

Figure A5b: Normal Q-Q Plot of Log(System size)

Figure A5c: Normal Q-Q Plot of Log(Maintenance effort)

Figure A5d: Normal Q-Q Plot of Log(Total effort)

Figure A6a: Scatter diagram plotting Percentage of COTS components with Development effort

Figure A6b: Scatter diagram plotting variables Percentage of COTS components and Maintenance effort

Figure A6c: Scatter diagram plotting System size with Development effort

Figure A6d: Scatter diagram plotting System size with Maintenance effort
Figure A6e: Scatter diagram plotting *Percentage of COTS components with Total effort.*

Figure A6f: *System size* plotted with *Total effort.*
Declaration

Whilst registered as a candidate for the above degree, I have not been registered for any other research award. The results and conclusions embodied in this thesis are the work of the named candidate and have not been submitted for any other academic award.
1.1 Introduction

The motivation for starting this research programme originated from two sources. The first was from working for a large multinational IT company, International Business Machines (IBM), where the focus of many software practitioners was in reducing the costs of developing and maintaining commercial software systems.

One software development method employed by the company involved custom building software systems from scratch. However, an issue with the custom-built method was with cost. As the complexity and size of systems being produced increased, in conjunction with the requirement to keep abreast of rapid changes in both hardware and software, so did the costs to develop systems and time required to bring the systems to market. According to Maurice Perks, IBM Fellow (personal communication, March 03, 2003) one reason for the increasing costs of the custom-built approach were greater salary costs for increasing numbers of software developers required to build larger, more complex systems.

In order for software companies, such as IBM, to remain competitive there was a requirement for them to increase productivity, whilst exploiting new technologies, reduce development costs, whilst producing systems in shorter time periods and to improve software quality in order meet customers’ expectations (Poulin, Caruso & Hancock, 1993).

The second motivating factor for conducting this research resulted from reading the literature on building software systems with Commercial-Off-The-Shelf (COTS) software components. One theme of the literature originating from the mid to late 1990’s focused upon the perceived cost savings of developing systems from COTS components and the design challenges of building systems using this method when system requirements cannot be satisfied from the available COTS components.

However, the focus of some later sources of literature, written in the early and mid 2000’s, suggested that although the development costs of COTS-based systems should
be lower than systems built using the custom-built approach the Total Cost of Ownership (TCO) of COTS-based systems could actually be higher than the TCO of custom-built systems due to higher costs to maintain COTS-based systems.

The change in focus found in later sources of literature may have occurred as a result of a realisation of higher costs to maintain COTS-based systems originally built in the 1990’s. Higher maintenance costs of COTS-based systems may not have been originally evident until the early COTS-based systems had been subjected to several maintenance cycles. When COTS-based design (CBD) was in its infancy the focus of attention of system developers was on the proposed development cost savings of this development method. The potential additional maintenance costs were not known (Clark & Clark, 2007). Several reasons were suggested for higher ongoing maintenance costs for COTS-based systems. Examples included component vendors dictating the evolution cycles of their products, thus compelling their customers to upgrade components in order to remain in support or of vendors withdrawing components from the marketplace because they deemed them no longer commercially viable, thus forcing customers to look for alternative solutions (Abts, 2002; 2004).

It was the volatility of the COTS component marketplace which was claimed to lead to increasing maintenance costs for COTS component customers because of their lack of control of the evolution of system components. This effect was more profound when dealing with multiple vendors because each vendor could adopt different product evolution cycles (Abts, 2002; 2004).

From a methodological perspective CBD was perceived as a solution to the spiralling costs of developing software from scratch (Abts, 2002; 2004; Albert & Brownsword, 2002; Baker, 2002; Ballurio, Scalzo & Rose, 2002; Yang, Bhuta, Boehm & Port, 2005; Li, Torchiano, Conradi, Slynstad & Bunse, 2006; Clark & Clark, 2007; Cook, 2007; Li, Torchiano, Conradi, Slynstad & Bunse, 2008). However, from the literature there was an indication that whilst CBD could lead to reduced development costs it was not without cost and the total cost of ownership of COTS-based systems could be higher than custom-built systems (Abts, 2002; 2004; Reifer, Basili, Boehm & Clark, 2003;
Therefore, organisations considering choosing the CBD method should be aware of the long term cost challenges associated with this method.

Thus, if an organisation’s decision for adopting the CBD approach is based on their perception that if CBD costs are less than those for custom-built systems then the total lifecycle costs of COTS-based systems will also be lower it may be too costly for them to change their minds once a system has been implemented.

1.2 CBS Functional Density Rule of Thumb

The CBS (COTS-based system) Functional Density Rule of Thumb (Abts, 2002; 2004) underpinned the reason why system developers started to use COTS components. The rationale for CBD was the idea that using COTS components helped realise cost savings by reducing the amount of effort required up-front to develop software systems. The logical inference was that the more COTS components used, the greater the overall development cost savings. However, Abts (2004) also suggested that the more COTS components used in a system, the greater the maintenance costs will be over the life of the system due to the volatility of the COTS marketplace. By volatility it was meant that component vendors controlled the evolution of their products, which ranged from dictating when new component versions were released through to withdrawing components altogether.

Abts (2004) suggested there can be seen to be conflicting notions of COTS-based system design: to minimise development costs, the best approach seems to be maximise the use of COTS components. However, to minimise maintenance costs the best approach seems to be to minimise the use of COTS components.

Abts (2002; 2004) developed the CBS Functional Density Rule of Thumb to help resolve this conflict. He suggested that to reduce system development costs the amount of functionality provided from COTS components should be maximised. However, to reduce maintenance costs this functionality should be provided from as few COTS components as possible.

Abts (2002, p. 5) articulated the CBS Functional Density Rule of Thumb as follows:
“Maxmise the amount of functionality in your system provided by COTS components but using as few COTS components as possible”.

1.3 Aim of the Research

The aim of this research programme is to challenge the perceived cost benefits of CBD by investigating the factors which influence the cost of building systems from COTS software components. If CBD is to be the cost-saving alternative to the custom-built approach to building software systems then the factors of this method should be fully understood to enable practitioners to make informed decisions on the most appropriate development method to use.

1.4 Research Objectives

The objectives of this research programme are as follows:

1) To assess if there is evidence to support, refute or extend the CBS Functional Density Rule of Thumb (Abts, 2002; 2004). From a design perspective, the CBS Functional Density Rule of Thumb recommends that to minimise development costs the greatest proportion of system functionality should be supplied from COTS components. However, to reduce maintenance costs this functionality should be supplied from the least number of components.

Further details on the CBS Functional Density Rule of Thumb (Abts, 2002; 2004) can be found in Section 1.2.

2) To understand the appropriateness of the Grounded Theory method within the Computer Science domain.

3) To understand the factors influencing the costs of COTS-based design.

1.5 Research Questions

The research questions for this research programme are as follows:
1) Is there evidence to support, refute or extend the *CBS Functional Density Rule of Thumb* (Abts, 2002; 2004)?

2) Are COTS-based system lifecycle costs significantly different from custom-built software systems and why?

3) What are the factors influencing the costs of COTS-based systems?

### 1.6 Research Plan

The plan of this research programme was as follows:

1) Literature review: the purpose of the literature review was to examine the literature on the use of COTS software components to build systems and to outline the main issues pertaining to CBD.

Details of the Literature Review are presented in Chapter 2.

2) Statistical Analysis: system data collected from IBM was analysed to determine if there was evidence to support or refute the perceived benefits of CBD, to establish if the total lifecycle costs of COTS-based systems were higher than those for custom-built systems and to confirm or refute the *CBS Functional Density Rule of Thumb* (Abts, 2002; 2004).

Details of the Statistical Analysis are presented in Chapter 4.

3) Grounded Theory: the Grounded Theory method was used to conduct a focused investigation of the factors influencing the costs of COTS-based systems using interview data collected from IBM practitioners and to identify evidence to support, challenge or extend the *CBS Functional Density Rule of Thumb* (Abts, 2002; 2004).

Details of the Grounded Theory analysis can be found in Chapter 5.

A diagrammatic structure of this PhD research programme is presented in Figure 1.1.
1.7 Contribution to Knowledge

The intended contributions to knowledge of this research programme were as follows:

1) To provide a better understanding of the issues affecting CBD.

2) To enable a better understanding of the approach, strengths and weaknesses of the Grounded Theory method.

3) To provide a better understanding about the quality of data required to separate different issues.

Figure 1.1: Diagrammatic structure of the PhD research programme.
1.8 Definition of Terms

For the purposes of this research programme the following definition of a COTS component was used:

“COTS components are sold, leased, or licensed by a vendor, supported and evolved by a vendor and used without internal modification by the customer (Meyers & Oberndorf, 2001, p. 126)”.

This definition is further explained in Chapter 2, Literature Review.

1.9 Conclusions

In summary, the aim of this chapter was to provide justification for the research programme, to detail the aim, research objectives and research questions, to provide details of the research plan, contributions of knowledge and definition of a COTS component.
Chapter 2 Literature Review

2.1 Introduction

The aims of this chapter are to examine the literature on the use of COTS software components to build systems and to outline the main issues pertaining to CBD. This constitutes the first phase of this research programme.

2.2 Definition of a COTS Software Component

Much of the literature relating to CBD either lacked concise definitions of COTS components or presented definitions which disagreed with other studies (Torchiano & Moriso, 2004). This point was supported by Bachmann, Bass, Buhman, Comella-Dorda, Long, Robert, Seacord and Wallnau (2000) who said that although the term CBD had been growing in popularity there was a lack of agreement among business analysts, researchers, technology producers and consumers about the nature of COTS components and how they could be used to design, develop and deploy new software systems. Furthermore, in some studies, terms such as ‘component’ and ‘product’, were used interchangeably when referring to CBD (Torchiano & Moriso, 2004).

Therefore, with regard to this study it was deemed important to ensure that an understanding of the meaning of all terms was provided. However, as with other complex subject areas it proved difficult to provide a simple, all encompassing definition of the terms relating to CBD.

2.2.1 Software Components

It is possible to consider COTS software components as belonging to the wider category of software components. Therefore, before looking at the specific nature of COTS software components features of general software components will be presented in the following sections.

Bachmann et al. (2000, p. 1) explained that the word ‘component’ was related to the word ‘compose’, which in turn originated from the Latin terms *com-* together and *ponere* to put.
The following statement linked the concepts *component* and *composition*. “One thing can be stated with certainty: components are for composition. Composition enables prefabricated ‘things’ to be reused by arranging them in ever new composites” (Szyperski, 1998, p. 3).

Composition embodies the essence of the ‘divide and conquer’ technique to reduce complexity and make large tasks manageable (Plakosh, Hissam & Wallnau, 1999). The traditional engineering disciplines employ this principle when building something large out of smaller parts.

Conceptually the notion of composition could be assumed to imply that all components fit together perfectly (Barbier, 2003). However, the integration of components is one of the main problem areas affecting CBD (Gamble & Gamble, 2007). This is because software composition relies on the interaction of software components. Successful component interaction assumes interoperability, which requires the sharing of common characteristics. If common characteristics are absent then components may not interact (Gamble & Gamble, 2007).

On defining the term *software component* Heineman and Councill (2001, p. 2) said “we found that few tasks more difficult than developing a concise, definite and clear definition”. A suggested reason for this was that etymologically speaking it is possible to view all software systems as being are built from components of some kind (Bachmann et al., 2000).

A further reason for the difficulty in defining a software component was because the word *component* has been used informally to mean different things to different people (Cheeseman & Daniels, 2001). For example, component definitions presented in publications aimed at non technical audiences could differ in complexity and detail to definitions found in technical documents and journal articles aimed at software engineers, programmers or academics. This can be demonstrated with the following two examples:

In Brown and Wallanau’s (1998, p. 38) high level definition a software component was classed as an “independent, replaceable part of an application that provides a clear distinct function”. Conversely, in D’Souza and Wills’s (1999, p. 387) more complex,
technical definition a software component was defined as, “a coherent package of software implementation that (a) can be independently developed and delivered, (b) has explicit and well-specified interfaces for services it provides, (c) has explicit and well-specified interfaces for services it expects from others, and (d) can be composed with other components, perhaps customising some of their properties, without modifying the components themselves”.

Another source of confusion over the definition of a software component was with the interpretation of scale: macro versus micro. Some definitions have adopted a macro-scale interpretation and considered a software component to be a complete application, package or product. Such an example could be a word processing or database package. At the other extreme other definitions of software components have taken a micro-scale view by comparing a software component to an object, as found within the object orientated domain (Beus-Dukic, 2000).

In relation to the last point a common issue found in the literature was of the terms object and component being used interchangeably. Szyperski (1998) suggested that this issue was due to historic factors linking the initial interest in software components of the early 1990’s to the growing popularity of the object oriented approach to software programming at the same time. As a consequence many people used the terms component and object to mean the same thing.

Further confusion over the terms component and object may have developed from the proposition that software components themselves were best constructed using an object oriented language (D’Souza & Wills, 1999). However, technically there is nothing to prevent software components from being written in any programming language.

The definition an object is beyond the scope of this chapter. However, it should be pointed out that although software components may exhibit similar characteristics to that of objects, such as information hiding, a software component should not be considered to be an object in the object oriented sense (Cheesman & Daniels, 2001). An object is at a too fine-grained level and should only be regarded as a possible building block for a component (Cheesman & Daniels, 2001). Thus, a software component can be made from objects but an object cannot be made from software components.
2.2.2 Interface

An interface is an integral part of a software component. A software component must possess an interface (Morisio & Torchiano, 2002; Egyed & Balzer, 2006). It is through an interface that components connect and communicate with other components or with other software items (Sametinger, 1997; Booch, 1998; Szyperski, 1998).

As with the term component there are various definitions of an interface. For example, the interface is a “definition of a set of behaviours that can be offered by a component” (Cheesman & Daniels, 2001, p. 8). The interface specifies what operations can be performed by a component (Meyers & Oberndorf, 2001). An interface is “a shared boundary across which information is passed” (Meyers & Oberndorf, 2001, p. 31).

Therefore, understanding the concept of an interface is integral to understanding the nature of a software component. If, for example, a component is defined as an independent piece of code (Brown & Wallanau, 1998) without an interface a component remains an independent piece of code and as such would be of little use. It is the interface which underpins the power of component based development facilitating the ability to connect together other independent pieces of code.

2.2.3 Code Reuse

Code reuse is a key concept of the definition of a software component. Code reuse embraces the notion that once a programming problem has been solved, or any other solution developed, it can be encapsulated within a component to be reused in different programming environments (Clemente & Hernandez, 2003; Vincent, Kink, Lay & Kinghorn, 2003).

Reuse is the focus of many definitions of a software component. For example, Sametinger (1997, p. 5) stated that “reusable software components are self-contained, clearly identifiable pieces that describe and/or perform specific functions, have clear interfaces, appropriate documentation and a defined reuse status”.

2.2.4 COTS Software Components

However, as with general software components, it is difficult to define COTS software components precisely. There are many different interpretations of a COTS component. This may be part due to the ever changing world of the COTS domain. Open systems
and the associated standards, web services and component frameworks have resulted in
the creation and deployment of COTS components in a variety of new contexts (Meyers
& Oberndorf, 2001).

One analysis of the term ‘COTS software component’ suggested that they exhibited
some of the same characteristics as standard software components, such as reuse and
defined interfaces (Place, 2001). However, two main differences between a standard
component and a COTS component were the added labels of ‘Commercial’ and ‘Off-the
Shelf’. However, there have been common misunderstandings with the use of the terms
‘Commercial’ and ‘Off-the-Shelf’. For example, for some people the word commercial
means that an item is available for sale while for others it means that the item has been
sold (Place, 2001). There was also confusion on whether or not an item was really off
the shelf (Place, 2001) For example, some COTS components require so much tailoring
by skilled staff that they cannot just be plucked from the shelf and implemented.

Clark and Clark (2007, p. 1) defined the ‘commercial’ aspect of a COTS component as
being “Sold, leased, or licensed for a fee to the general public”. Meyers and Oberndorf,
(2001, p. 126) expanded on the commercial characteristic, defining a COTS component
as being, “offered by a vendor trying to profit from it and supported and evolved by the
vendor, which retains the intellectual property rights”. It can be seen from both
examples that commercial is aligned to the development and sale of components by a
vendor for profit, rather than with the case of standard software components which may
be developed for internal use by an organisation.

The ‘Off-the-shelf’ aspect of COTS was defined as “Available in multiple, identical
copies” (Meyers & Oberndorf, 2001, p. 126). The analogy being with a grocery
supermarket where multiple copies of different products, such as tins of baked beans or
packets of soap, are available for purchase off the shelf by the general public.
Furthermore, a common trait of COTS components was a general-purpose piece of
software which was not developed for specific projects (Torchiano & Morisio, 2004).

The term COTS product has been used in some COTS classifications. A COTS product,
for example, was considered to be an entire application performing a self-contained set
of tasks. Word processing, spreadsheet and electronic mail applications were examples
of COTS products. This differed from a COTS component which was defined as a
smaller software entity, which when integrated with other COTS components an application (Allan, 2003).

Morisio & Torchiano (2002) classified all COTS software components as COTS products. They proposed a classification based upon COTS product size. For example, a small COTS product was less than 0.5 Mega Bytes (MB), such as Microsoft Chart Control; a medium COTS product was between 0.5MB and 2MB; large, between 2MB and 20MB, for example, Samba and huge, a product such as Windows XP, was over 20MB.

However, in this thesis the terms COTS component and COTS product were considered to mean the same thing.

A variation on the COTS component theme was the business component. A business component is a unique implementation of the business logic, rules and constraints of a particular type of business that has been captured into executable code. It can be defined as follows: “a self-contained software entity with well-defined interaction points facilitating the accessing and execution of a coherent package of functionality” (Barbier & Atkinson, 2003, p. 6).

The similarity between a business component and a COTS component is the notion of the interface. In a business component the interface was referred to it as an ‘interaction point’ (Barbier & Atkinson, 2003). However, a business component tends to be self contained, capturing large chunks of a business operation. However, for a general business component to emulate specific customer requirements it must either be configurable, which contests the black box nature of a COTS component, or be developed specifically for a customer, which contests the idea that a COTS component should be available in multiple, identical copies, as defined earlier by Meyers and Oberndorf (2001).

2.2.5 Black Box Nature of COTS Software Components

The black box nature was a common focus of many definitions of COTS components. Black box implied that the source code and inner workings of a COTS component were not made available to the consumer (Kotonya, Onyino, Hutchinson, Sawyer & Canal, 2003).
The concept of a ‘black box’ within the COTS domain implies that a software component satisfies a set of functions; how it does this is not important as the component’s inner workings are not made visible to the end user. However, some larger components may require some form of configuration which does not involve modification of the source code (Allan, 2001).

Figure 2.1 is a diagrammatical representation of a black box component. It accepts inputs, processes the data then produces output.

![Diagram of COTS Component](image)

**Figure 2.1: A black box approach to COTS components (Allan, 2003).**

There are several benefits to the black box approach. For example, rendering a component’s source code unavailable to end-users enables COTS component vendors to protect their intellectual capital. This is can be seen to be an important advantage if a vendor’s primary goal is financial profit. Another benefit of the black box approach is that a component can potentially be developed from any programming language. Furthermore, the effect of an upgraded version of a component, which does not supply additional functionality, should be transparent to the end user because only the source code is changed.

A predicament of the black box concept is that customers need to have faith that a COTS component performs its documented functions. When source code is not available for scrutiny it is difficult to establish how a component works, its performance and security implications and whether it contains any bugs (Gamble & Gamble, 2007).

Further problems may arise if the components do not meet the exact requirements of the system. If the source code is not available for modification one solution is to develop custom ‘wrapper’ or ‘glue’ code, which may externally modify the component’s
functionality. However, this can add significant costs to a CBD project (Reifer et al., 2003; 2004).

There are variations to the black box nature of COTS components. For example, concepts of grey, opaque or white box components have been proposed which allow for varying degrees of customer configuration (Bachmann et al., 2000). However, these terms rarely implied unrestricted access to the component’s source code. Acquisition of white and grey box COTS components may result in a more flexible set of components, although the trade-off may be the need of specific configuration skills by the customer or end-user (Bachmann et al., 2000).

2.2.6 Glue Code and Wrapper Code

Glue code is the term which describes code developed to enable COTS components to operate together, with existing legacy code or other system parts (Looney, 2002; Yang et al., 2005). For example, given two components which need to be integrated within a system, but which do not support the same architectural standards, glue code may be created to modify the output from one component making it compatible for processing by the other component.

The purpose of wrapper code is to create constructs between or around COTS components which hide the implementation details beneath a stable interface. Wrapper code can provide a buffer against vendor changes in either the component’s interface or functionality. This is done for several reasons:

• to refine incompatible, poorly designed or complicated interfaces;
• to allow incompatible components to work together (for example, where components produce incompatible data formats or are built to different architectural standards);
• to enable portability of components whereby the wrapper code can function on different operating platforms.

Wrapper code can be implemented in different ways. One example is where a ‘layer’ of code is created which translates one or more components’ existing interfaces into a compatible interface. In such cases, the wrapper code may provide an interface between components and an operating system or between the target application and the network.
transport mechanism. Another example involves creating custom wrapper code for each component, whereby the responsibility for manipulating the output of a component is performed by the wrapper code, thus isolating the component’s interface from the user (Dean & Li, 2002; Looney, 2002; Shin & Paniagua, 2009).

One recommendation was for all COTS components to be wrapped to enable all parts of a COTS-based system to function consistently. All interactions between components were performed through wrappers, glue code providing the interaction between components (Sassi, Jilani & Ghezala, 2003). However, a problem with this approach was that the wrapper and glue code may have to be modified every time a component is changed or upgraded. Depending upon the number of components this could result in a lot of additional effort.

2.2.7 COTS Component Definition used in this Thesis

With reference to the above discussion the following definition of COTS components will be used in this thesis. This definition was chosen because it encompassed the concept that COTS components are produced by vendors, who tend to control the evolution of components. Furthermore, although some COTS components can be configured by customers vendors do not normally provide access to the component’s source code.

“COTS components are sold, leased, or licensed by a vendor, supported and evolved by a vendor and used without internal modification by the customer” (Meyers & Oberndorf, 2001, p. 126).

2.3 Conceptual Justification of COTS Software Component Use

Within software engineering CBD has been gaining popularity when developing software systems (Abts, 2002; Albert & Brownsword, 2002; Baker, 2002; Ballurio et al., 2002; Abts, 2004; Yang et al., 2005; Li et al., 2006; Clark & Clark, 2007; Cook, 2007; Li et al., 2008). This involves producing a software system by integrating software components which have been previously developed, produced, supplied and licensed by component vendors.
Building ‘things’ with components is well established in mechanical, electrical and civil engineering disciplines (Szyperski, 1998; D’Souza & Wills, 1999). According to Maurice Perks, IBM Fellow (personal communication, March 03, 2003), componentry is a natural engineering technique that has been utilised by engineers since the onset of the industrial revolution. Early engineers realized that it was impractical to build large constructions, such as a bridge or factory, as a single part. However, the component concept encompassed more than just building something large from smaller parts. Individual components were normally produced separately then assembled according to a prepared design. For example, to build a steam engine, the engineers did not start by designing each nut and bolt from scratch. (D’Souza & Wills, 1999). Maurice Perks, IBM Fellow (personal communication, March 03, 2003), suggested that it may be more accurate to view Robert Stephenson, of ‘Stephenson’s Rocket’ fame, as a ‘component integrator’ rather than inventor. He integrated existing steam boiler technology with a variety of iron and steel ‘components’ to produce his famous locomotive.

There are examples of component use within other fields of engineering: Henry Ford’s successful method of constructing motor cars out of steel and automotive components (Heineman & Council, 2001); the assembly of electrical components to produce televisions, washing machines and other electrical appliances (D’Souza & Wills, 1999) and within civil engineering the use of prefabricated brick, window, door, roof tile, timber joist and electrical and plumbing components to construct buildings (Jacobson, 1993).

An analogy can be seen between methods used by software engineers to produce COTS-based systems and the component approach to construction which originated from traditional engineering disciplines. With CBD systems are constructed from prefabricated software components. Thus, the COTS software component can be compared with iron girders used by civil engineers or electrical components used within electrical engineering (Jacobson, 1993).

However, there are some exceptions to the above analogy. The results of traditional engineering activities are some sort of physical construction. Examples of civil engineering end products are a bridge or motorway. However, with CBD tasks the results are not final products, but rather blueprints for products. It is the execution of the
final software which produces the product, such as a report; printed output; financial cost calculation and so on (Szyperski, 1998).

Another difference between CBD and other engineering disciplines concerns natural laws. For example, when designing a bridge the civil engineer must consider the forces of gravity as well as the load-bearing qualities of steel and other materials. According to Maurice Perks, IBM Fellow (personal communication, March 03, 2003) the same laws do not apply when building software systems.

However, the purpose here is not to discuss the merits or failings of CBD when compared with other engineering disciplines. The important point is that conceptually, building systems with COTS software components requires design and planning stages which are similar to those employed within traditional engineering.

2.4 Historical Context of using COTS Software Components

The use of COTS software components to build systems is a relatively new concept. Szyperski (1998) stated that what he called ‘componentry’ was gaining recognition as a key technology for the construction of high-quality, evolvable, large software systems in timely and affordable manners. CBD represented an emerging development paradigm (Kotonya et al., 2003); CBD has been gaining more attention from both research and industrial communities (Torchiano & Morisio, 2004) and processes are increasingly moving away from the time consuming composition of custom software to the integration of COTS components (Yang et al., 2005).

However, Maurice Perks, IBM Fellow (personal communication, March 03, 2003) said that he was involved in experimenting with ‘COTS-type components’ in the early 1980’s. Heineman and Councill (2001), quoting Ivar Jacobson, explained that he worked with software components more than thirty years ago, using them to build a large successful system for the telecommunications industry. In the 1960’s Brooks (2002) proposed concepts which are now central to CBD: Buy before Build and Reuse before Buy. He suggested then that software reuse was the key behind reducing both development time and costs; why waste time creating software from scratch when someone else has already done it!
The invention of the silicon chip and integrated circuitry led to a revolution in computer hardware development; reusable hardware components. The net result enabled silicon chips to be manufactured by numerous companies and the ensuing market forces helped drive the cost down, thus also reducing the cost of computer hardware (Cox, 1987). The inception of COTS had similar roots and included the notion of software warehouses supplying components which would seamlessly integrate with existing programs and other COTS components. COTS components were envisioned as the software equivalent to the silicon chip (Cook, 2007).

The advent of the open systems approach to software engineering was also a factor in the growth of CBD (Meyers & Oberndorf, 2001). An open system was defined as a collection of interacting software, hardware, and human components, designed to satisfy stated needs, with the interface specification of components which are: fully defined, available to the public and maintained according to group consensus (Oberndorf, 2007). One result of the open systems approach was the development of common architectural standards. Thus, the vision was that components would work with any system supporting the same standard.

One aspect of this definition was the specification of common interface standards which were adhered to by COTS component vendors (Bachmann et al., 2000). This enabled compatible components from one supplier to be substituted by those from other suppliers (Oberndorf, 2007). However, it should be noted that COTS components themselves are rarely freely available to the public (Meyers & Oberndorf, 2001).

The aspect of standards presents an important point in the evolution of the COTS component approach to software engineering. Prior to the advent of Open Systems standards software systems tended to be propriety in nature. For example, IBM was a major player in the growth of the mainframe system. The operating system (O/S360, OS/370 and later the OS/390) and many of the applications which ran on these systems were developed, sold and maintained by IBM. It was difficult for non-IBM developers to provide applications which would run on these systems because IBM owned the intellectual capital for this technology. Even IBM’s version of UNIX, AIX, contained some propriety modules. The growth of open systems marked a major deviation from propriety systems. Visible source code and openly documented standards provided the opportunity for any developer to create programs and components to run on any system
conforming to the same standards. Thus, the software version of the ‘nut and bolt’ could potentially be created by anyone rather than by the ‘owner’ of the propriety system (Boulanger, 2005).

2.5 Perceived Benefits of COTS-Based Software Engineering

To gain competitive advantage there is a requirement for software companies to develop software systems quickly and cost-efficiently. The time and cost in which a system can be brought to market, and then modified to reflect different business opportunities, can have a direct bearing on the profitability of a company (Bhuta, Boehm & Meyers, 2005; Li et al., 2006; Lin, Lai, Ullrich, Kuca, McClelland, Shaffer-Gant, Pacheco & Dalton, 2007).

The perceived benefits of building systems from COTS components, compared with custom system development, were lower development costs and reduced development time and reduced maintenance and evolution costs (Polze, 1999; Heineman & Councill, 2001; Chung & Cooper, 2002; Seacord & Wrage, 2002; Albert & Brownsword, 2002; Clemente & Hernandez, 2003; Kotonya et al., 2003, Bhuta et al., 2005; Li et al., 2006; Lin et al., 2007; Boehm & Bhuta, 2008; Finnegan, 2008; Postmus & Meijler, 2008; Sheng & Wang, 2008; Suleiman, 2008; Tomita, Fujiwara, Kawasaki, Miwa & Nagai, 2008).

Economic cost considerations have enticed many organisations to build software systems with COTS components (Bhuta & Boehm, 2007; Boehm & Bhuta, 2008).

The advantage of the component approach to engineering was that engineers can benefit from the reduced costs of reusing existing component designs, compared with the increased costs and effort involved in ‘reinventing the wheel’ for each successive implementation.

The assumption held by some practitioners was that the quality of COTS components should be higher than custom software artefacts because they have been used and tested by other customers (Perez, Ramos & Carsi, 2008). However, this may only be true when the customer base of components is large enough to provide sufficient testing. In
situations where the customer base is small vendors may not be able to benefit from customer testing of their products.

CBD was seen as offering a structured way of achieving effective code reuse because the development, testing and validating work of each component had been performed by the COTS component vendor (Clemente & Hernandez, 2003; Vincent et al., 2003).

A main driver behind the initial interest in COTS software components was the spiralling costs of developing large bespoke software systems. The ever increasing importance and reliance on more complex information systems, coupled with the extortionate costs of developing customer specific systems resulted in the necessity of alternative, cheaper and more flexible software development paradigms (Allan, 2003). Furthermore, during the 1980’s the major costs for developing systems shifted from hardware to software. CBD was a proposed method to reduce the cost of developing software systems (Horowitz & Lambert, 2006).

Taking this point further the Software Group (2002) provided the following statistics on the success of some important software projects:

- At best, 5% of ALL software projects were finished on time and within budget.
- Less than 1% (perhaps 1 in 150) of all software projects were completed on-time, under-budget and to the precise needs of users.
- 75% of the software projects observed were never completed or they were finally completed too late to be of practical or economically feasible use to the organization.
- The average cost estimation of software projects was so poor that the actual cost was up to three times the original prediction.

Although the above statements are vague and lack detail of the type of software projects and the number of projects sampled they provide an interesting insight into the main concerns of both the software producer and customer. These were the cost and time for new software systems to be developed and the quality of the completed system.
2.6 Issues with COTS-Based Development

Numerous issues were identified with building systems from COTS components. The aim of the following sections is to present the main issues affecting COTS-based development.

2.6.1 Development Approach

An important issue regarding CBD was the requirement for a different design methodology, compared with other ‘traditional’ methods used for developing software systems. This was described as a paradigm shift; the essence being that the system engineer changes from being a producer of software to a consumer and integrator of COTS components (Meyers & Oberndorf, 2001; Boehm, Yang, Bhuta & Port, 2005; Finnegan, 2008; Yang, He, Li, Wang & Boehm, 2008). Furthermore, different parts of systems built from COTS components may evolve in different ways, indicating that no single development process should be used (Boehm, 2010).

Examples of traditional system development methods include iterative development methods, Prototyping, Rapid Prototyping, Unified Development Processes, Agile development methods and the Waterfall Model (Miller, 2008; Yang et al., 2008; Boehm, 2010; Woodward, Surdek & Ganis, 2010).

The Waterfall Model, for example, commences with a requirements phase, followed by architectural selection and system design, system implementation and deployment and concluding with the testing and maintenance phases. The assumption was that each phase was performed in sequence.

For the purpose of this study requirements are anything that drives design choices (Finnegan, 2008).

The problem of applying the Waterfall Model to CBD was that after determining the system requirements it may be difficult to locate COTS components which satisfy the specified customer requirements and are compliant with the desired architectural model. With this in mind some researchers reported that the use of the traditional software development approach rarely worked for COTS-based systems (Albert & Brownswod, 2002; Ballurio et al., 2002). More emphatically, adopting a ‘waterfall’ implementation
to building a system using COTS components would probably fail because the Waterfall Model does not account for the volatility of the COTS marketplace (Bhuta, et al., 2005).

The features of CBD, compared with the Waterfall Model, were epitomised by the following characteristics: Development essentially involves combining existing software products; the marketplace exerts a strong influence on product availability and continuous tradeoffs between system requirements and component availability occur (Torchiano & Morisio, 2004).

With reference to the above points a cyclical development paradigm was identified as being far better suited for CBD than, for example, the waterfall approach (Meyers & Oberndorf, 2001; Place, 2001; Boehm et al., 2005). A cyclical development approach involved the continuous iterative execution of requirements definition, architectural selection, component identification and market research phases until system completion. Because each phase could be quickly implemented such an approach addressed the constant trade-off between system requirements and the availability of suitable components.

A further benefit of the cyclical development approach is that feasibility of a COTS-based solution can be determined early on in a project (Yang et al., 2005). For example, with the absence of suitable COTS components the choice to abandon the project or to adopt a custom-based approach can be identified early on in the development timeline. With the waterfall model this outcome would occur much later in the system lifecycle, thus with excessive time being wasted.

Other cyclical development models have been proposed. Boehm (1988) termed his methodology a ‘spiral model’. This model is partitioned into four major stages:

- Planning;
- Risk analysis and mitigation;
- Engineering;
- Customer evaluation.
Commencing with the *Planning* stage, an initial cycle of the different stages should result in an early indication of the feasibility of using the COTS-based approach. Once feasibility is assured, further iterations of all stages advance system development through to implementation (Boehm, 1988).

An alternative version of the spiral development model was proposed by Albert and Brownsword (2002). They termed this the *Information Technology Solutions Evolution Process* (ITSEP). The focus of this model was the simultaneous execution of the following four development phases (known as *spheres of influence*): The Marketplace; Architecture and Design; System Requirements and Programmatic Risk. Additionally, continuous negotiation between customer and software engineer throughout the development of the system was deemed paramount in determining the functionality of the system.

There are similarities of the spiral development approach to the concept of prototyping Tracz (2001). With both prototyping and the spiral model during the first iteration several software development phases can be performed in close succession. Following component acquisition, part of the system can be built, tested and shown to the customer. A benefit is that the customer can see a subset of the finished system early on.

When employing a spiral development model to build COTS-based systems the question on whether system requirements should drive the component selection or, conversely, whether component availability should drive the system requirements and architecture has been posed by researchers (Oberndorf, Brownsword, Morris & Sledge, 1997). There appeared not to be a simple answer to this question. However, projects not allowing for flexibility of system requirements are not ideally suited for a CBD approach. This is because the source of COTS components, the COTS marketplace, can change continually.

**2.6.2 COTS Component Selection and Acquisition**

A limiting factor of building systems with COTS components concerns the availability of suitable components. COTS component suitability has been expressed in two ways:

1) Do COTS components which support the requirements of the system exist in the marketplace (Carney, Place & Oberndorf, 2003)?
2) Will the identified COTS components work together (Carney et al., 2003)?

It can therefore be concluded that if points 1 and 2 cannot be satisfied the COTS-based approach may not be viable for a project.

Related to the above point, in projects where the selection process was only limited to an evaluation of component functionality, rather than to also consider issues of component interoperability, the result tended to be increased costs and schedule overruns (Bhuta & Boehm, 2005; Boehm & Bhuta, 2008).

Furthermore, challenges arise with the identification of suitable components from vendor repositories. Conventional retrieval methods may not evaluate the total characteristics of a component, because they tend only to consider a single aspect of the component or require an additional description (Washizaki & Fukazawa, 2001).

The problem with locating suitable COTS components in the commercial marketplace stems from vendors, in attempting to maximise profits, developing components to appeal to as wide a customer base as possible. The net result of this approach is that many components are built independently of any specific system requirements (Place, 2001).

Furthermore, COTS software vendors are more likely to produce components which reflect current commercial strategy, will capture a significant market share and will run on current technologies (Carney et al., 2003). Therefore, when customer requirements do not adhere to these factors the likelihood of identifying suitable components can be reduced.

2.6.3 Coupling and Cohesion

Coupling and cohesion are important terms in computer science. When applied to CBD they help determine COTS quality and the ease in which COTS components can be integrated and replaced (Cook, 2007).

Coupling refers to how components interact. There are various levels of coupling. For example, Message Coupling is the loosest type of coupling. Components are not
dependent on each other. A public interface is used to pass messages between them. Conversely, Content Coupling occurs when one component relies on the internal workings of another component. For CBD Content Coupling does not facilitate easy integration and replacement of components because one component may have a strong association with other components. This makes it difficult to remove or replace one component independently of other components (Cook, 2007).

For COTS-based projects the lowest level of coupling is recommended, where the actions of one component have little effect on other components. Low coupling, such as Message Coupling, gives COTS the *plug and play* ability in terms of a standard interface. This gives the ability to replace one component with another (Cook, 2007).

Cohesion relates to the modularity of a system and is a measure of how COTS components perform a single task. In a well modularized system, routines are reasonably small and perform one action (Cook, 2007).

As with coupling there are various levels of cohesion. The highest and recommended level, Functional Cohesion, which is when parts of a component are grouped because they contribute to a single, well-defined task. In contrast, the lowest and worst level of cohesion, Coincidental Cohesion is when parts of a system are grouped arbitrarily. The parts have no significant relationship (Cook, 2007).

### 2.6.4 Component Selection

There are numerous COTS component selection analysis methods (Seacord, Hussam & Wallnau, 1998; Bhuta & Boehm, 2005; Boehm & Bhuta, 2008; Sheng & Wang, 2008). For example, the Compatible COTS Component Selection Method (Bhuta & Boehm, 2005) was proposed as a suitable method for filtering out incompatible components and for determining compatible sets of COTS components. The main feature of this method was to classify candidate components into function groups which helped developers to assess component capabilities and interoperability. However, this study was limited to a dozen small to medium e-services projects. Further work was required to determine how well the method scaled up to include projects containing more function groups.

The COTS Software Selection Process (Lin et al., 2007) recommended that COTS component evaluation was performed early on in a project, following the requirements
gathering phase. By not proceeding to the design phase before choosing COTS components facilitated an open-minded approach to component evaluation which could assist in the identification of new technologies and capabilities. It was proposed that this approach resulted in improved design and business practices.

Details of other COTS component selection analysis methods have been published (Cechich & Piattini, 2007). However, although benefits of the methods were proposed by the authors, in practice there were limitations of the various methods. For example, for some of the methods, entering search criteria produced misleading results because decisions on component suitability were dependent upon further in-depth analysis of interface compatibility (Cechich & Piattini, 2007).

However, several ‘Lessons Learned’ were identified which should be considered when selecting COTS components (Cechich & Piattini, 2007):

- An assessment of the functional suitability of candidate COTS components should be performed early on in a project, facilitating unsuitable components to be discarded early on. This process allowed for any remaining components to be subjected to more focused analysis of suitability. The detailed analysis of component suitability can be time consuming. Thus, the claimed benefit of this approach was of less time wasted on considering unsuitable components.

- The assessment of COTS vendor reliability and maturity was as crucial as the identification of suitable components. This point was deemed important because components may be withdrawn from the market by COTS vendors or vendors themselves cease trading, leaving components unsupported.

- The functional descriptions of COTS components should adhere to a common standard. This requirement was deemed important to enable the detection of candidate components. The viability of this approach would need the cooperation of COTS vendors.

Research has been conducted into classifying component characteristics (Bhuta & Bohem, 2007). For example, Bhuta and Bohem (2007) developed an Interoperability Assessment Tool which attempted to classify and categorise the interoperability
attributes and characteristics of components. The attributes were grouped into the following:

- COTS general attributes;
- COTS interface attributes;
- COTS internal assumption attributes;
- COTS dependency attributes.

However, they were reliant on vendors publicising component attribute information.

It can be seen that a requirement underpinning the success of component characterisation assessment tools is two fold:

- Populating it with correct information from as wide a range of components as possible;
- Keeping it current, in response to the number of new components being released onto the market.

Failure to perform the above could soon render a tool out of date and providing limited coverage of available components.

Selecting components which adhered to the same architectural standard was also deemed beneficial as there was a greater chance that the components would function together with minimal modification (Simanta, Lewis, Morris & Wrage, 2008).

Standards are a way of reaching agreement among interacting participants. A standard establishes uniform engineering or technical specifications, criteria, methods, processes or practices (Simanta et al., 2008).

Standards can exist at different levels. For example, at the syntactic level, components can programmatically interact by understanding the syntactic meaning of the data and control commands. CORBA and COM are examples of syntactic standards. At the semantic level components can interact by either following the same semantic data model or by translating between different semantic data models (Simanta et al., 2008).
However, in many cases, the documentation accompanying COTS components either failed to fully describe component functions or described their uses incorrectly (Chang, Mariani & Pezze, 2009). The result was that COTS system designers and developers may inadvertently select unsuitable components. It can be seen that this issue was more likely to occur if technical authors are based in countries where English is not their primary language.

### 2.7 Maintenance of COTS-Based Systems

There appeared to be a lack of consensus on what constituted the maintenance of COTS-based systems. One recommendation was that maintenance should include any software lifecycle activity occurring after system implementation (Reifer et al., 2003; 2004). However, in situations where components require upgrading during the system development phase it is not clear if this should be classified as a development task or a maintenance task.

To aid the clarification of what constituted maintenance candidate tasks were divided into two groups (Reifer et al., 2003; 2004): The first group was named ‘Sustaining Engineering’ and included activities which were designed to sustain software operations and support activities. These activities included: Application repairs, configuration management, quality control, security administration, user support and training, continued COTS package evaluation, COTS vendor liaison and COTS component patch management.

The second group was called ‘New Version Release’ and included activities related to upgrading components, incorporating new system features and functions and scheduled repairs. Examples of these activities were: version development, beta testing and documentation writing, COTS component repair, tailoring and integration.

The long term maintenance of COTS-based systems was identified as an area where little expertise existed. As such, system maintenance and evolution tasks could become the most expensive stages of the system life-cycle (Carney, 1997). In some instances it was found that COTS-based systems were more costly to maintain than comparable custom built systems (Clark & Clark, 2007).
The above statements were related to several factors. Firstly, few software life cycle models adequately addressed the maintenance of COTS-based systems. For example, guidance on which activities need to take place during the maintenance phase of a COTS-based system was omitted (Reifer et al., 2003; 2004). Secondly, unlike custom built systems where maintenance is normally performed at the source code level, the maintenance of COTS-based systems typically involves dealing with a set of black boxes for which the source code is invariably owned by third party vendors (Vigder & Kark, 2006). Assessing of the effect of upgrading a black box component can be difficult and costly (Clark & Clark, 2007). Thirdly, this factor can be compounded because component vendors tend to dictate the upgrade schedule of their components. Thus, it can be difficult for system maintainers to adequately plan for successful component refresh activity if they have not been made aware of component upgrade requirements and the dependencies with other system parts. This issue can increase exponentially as the number of COTS components increases because each component may have different upgrade schedules and dependencies.

The uncertain nature of the COTS domain was identified as an area of concern for system maintainers (Brownsword, 2000; Li et al., 2008). To clarify the uncertain nature of the COTS domain Oberndorf, Brownsword and Sledge (1997) identified the following eight characteristics of the COTS marketplace:

- The COTS marketplace changes continuously;
- Commercial pressure, not individual system requirements, drives COTS component production;
- There may be built-in assumptions by vendors about how the COTS components should be used which may not be appreciated by the end user;
- Licensing and intellectual capital rights may permeate many COTS components;
- Vendors, not users, control the frequency and content of COTS component upgrades;
- End-users tend to have limited visibility to component source code;
- Vendors may base components on architectural assumptions that vary across different components;
• Components may have interdependencies which manifest themselves when components are changed.

In addition, the market-related instability may lead component providers to terminate maintenance support completely (Li et al., 2008).

One theme permeating the above list was the lack of control which a software engineer has over the COTS domain. With the custom-built approach the software engineer was able to build and maintain a system with reference to known specifications. However, due to the ‘black box’ nature of many COTS components issues of unpredictability may advertently be introduced when individual components are upgraded or changed as a result of maintenance. Furthermore, a software engineer has to contend with many other areas of flux which were not suited to custom approach to maintaining systems.

2.7.1 Factors Influencing the Cost of Maintaining COTS-Based Systems

Compared with the maintenance of custom-built systems several factors were identified to impact the cost of maintaining COTS-based systems (Clark & Clark, 2007). The following are examples of these factors:

• Licensing: In addition to the licensing costs the administration of component licence agreements and support contracted required effort. This source of cost was not required for custom-built systems (Clark & Clark, 2007).

• Evaluation of new component releases: The effort required to evaluate and understand the implications of upgrading a component. The variables deemed to influence maintenance cost were workload requirements to investigate: component interactions with other COTS components or custom code; glue code or wrapper code modifications requirements; and component reconfiguration requirements (Reifer et al., 2003; 2004; Clark & Clark, 2007).

• Identification of software defects: The identification and resolution of component defects could take time and effort. With COTS-based systems vendor support was normally required. Furthermore, added time and cost could be incurred if the vendor refuted the source of a software defect.
However, with custom-built systems the source code can be made accessible to examination when defect diagnostics is required (Clark & Clark, 2007).

- **Number of Components:** The number of components integrated within a COTS-based system was identified as a significant variable influencing maintenance costs (Abts, 2002; 2004; Reifer et al., 2003; 2004; Clark & Clark, 2007). Furthermore, maintenance complexity and costs would increase exponentially as the number of independent COTS components integrated into a system increased (Reifer et al., 2003; 2004). This was because the maintenance of each component incurred a cost in effort and time. Therefore, the maintenance costs of numerous components would be much higher. These factors were exacerbated when a system included components supplied by more than one vendor as there may be a dependence on the various release cycles of the different COTS vendors.

An additional caveat on the number of components used to build a system was added by Abts (2002; 2004). He explained that a conflict existed. The greater the proportion of a system constructed from COTS components the lower the cost of initial development. However, the more COTS components used in a system the higher the maintenance costs throughout the system’s life. Abts (2002) stated that in order to reconcile these views the proposed approach to minimise both development and maintenance costs involved increasing the percentage of system functionality delivered via COTS components, rather than increasing the number of COTS components used in a system.

To address these issues the *CBS Functional Density Rule of Thumb* was proposed:

> “Maximize the amount of functionality in your system provided by COTS components but using as few COTS components as possible” (Abts, 2002, p. 5).

**2.7.2 Strategies for Reducing Maintenance Costs**

The architectural design of a COTS-based system has a significant impact on its maintainability (Vigder & Kark, 2006). However, the implied issue with this statement is that can be very difficult, and expensive, to add *maintainability* properties after system implementation. Problems relating to system maintenance may not become clear until system maintenance activity is required. However, at this time it may be too late to
make architectural changes. Therefore, how a system is to be maintained should be considered during the early stages of a system’s development (Vigder & Kark, 2006). The following design considerations were deemed to be important:

Data cohesion: The aim is for COTS components, which together perform distinct, tasks to be encapsulated within a single object, known as a mediator. The reported benefit to system maintenance was that a component could be replaced without affecting system stability because this change was invisible to the rest of the system. This was also termed as information hiding and could be implemented with glue or wrapper code. A disadvantage of this approach was the requirement for additional effort, to be deployed during the development phase, to define the architectural configuration of each mediator. If the motivation of choosing the COTS approach was a faster time to market then additional time spent during system development could challenge the planned implementation schedule (Vigder & Kark, 2006; Clark & Clark, 2007).

Controlled interfaces: A design consideration, which was proposed to minimise the negative effect of changing a component was to use accepted standards for interfacing COTS components. Therefore, the use of proprietary vendor interfaces is not recommended unless all system components are to be supplied by the same vendor (Vigder & Kark, 2006).

Minimal component coupling: The more dependencies between components, the greater the coupling and the more difficult component maintenance becomes because making a change to one component could affect its interaction with other parts of the system. Therefore, good system design should aim to reduce component coupling down to a minimum. This could be achieved with wrapper code or the use of middleware products (Vigder & Kark, 2006).

Other strategies which are not directly related to system design should also be considered. The ease in which COTS-based systems can be maintained can be helped if good working relationships exist between software engineers and COTS component vendors. Based upon the upgrade schedule the component user must then decide if and when to upgrade the components. There may be risks associated with either course of action. For example, new versions of components may fail to integrate properly with
existing components. Conversely, failing to upgrade may result in ‘out of support’
components, which in turn may compromise the system’s integrity.

However, it may not be possible to determine the quality of long-term support that a
component supplier will provide, how long a commercial vendor will stay in business or
the contingency options if a vendor stops supporting key components or ceases trading
(Carney, 1997; Gluch & Weinstock, 1997).

2.8 Challenges to the Perceived Benefits of CBD

The perceived benefits of using COTS components have been widely publicised.
However, the risks have often been overlooked (Place, 2001). For example, the quality
of COTS-based software does not always measure up to expectations (Ballurio et al.,
2002) and that CBD can pose many problems for organisations intending to adopt it
(Onyino, Kotonya, Hutchinson & Sawyer, 2002; Kotonya et al., 2003).

Maurice Perks, IBM Fellow (personal communication, March 03, 2003), drawing on his
experience of many years involvement with IBM’s software strategy planning, proposed
slightly tongue and cheek ‘Perk’s 37th Law’: “If things are not designed to fit together
they won’t!”

One main challenge to the perceived benefits of building systems from COTS
components was that of cost. Although many sources from the literature presented a
reduction in system development costs, compared with custom development, as a
benefit of CBD (Bhuta et al., 2005; Li et al., 2006; Yang et al., 2008) an overriding
impression given to the reader was that the total cost of ownership would also be less
for COTS-based systems compared with custom built systems.

A proposition found less often in the literature was that due to high maintenance costs
the total lifecycle costs of COTS-based systems could be higher than custom built
systems (Reifer et al., 2003; 2004). Thus, during their life the costs to maintain COTS-
based systems could outweigh any cost savings achieved during the initial development
phase (Abts, 2002; 2004).
2.9 Conclusions

The review of the literature has highlighted numerous issues with the use of COTS components. On one hand building systems with COTS components were perceived to provide a ‘silver bullet’ solution to the cost challenges of building systems from scratch.

However, some of the early literature on CBD only discussed the perceived benefits of reducing developments costs. This could be explained with reference to the length of time taken for systems to mature to the point where the full extent of maintenance costs could be realised.

However, in some of the literature it was seen that doubts on the long-term cost benefits of CBD started to emerge. Specifically, that there were indications that ongoing maintenance and support costs of COTS-based systems could be higher than those of custom developed systems, thus overshadowing any reductions in the development costs of CBD.

Therefore, if CBD is to live up to its aspirations as a method of reducing costs then the factors affecting total system lifecycle costs of using this method need to be understood.

The objectives of the following sections of PhD research programme are to investigate the factors affecting the cost of CBD.

The aim of the next chapter is to define the design of the research programme which will be used to address the programme objectives.
Chapter 3 Research Methods

3.1 Introduction

The aim of this chapter is to provide details of the research methods used in this research programme.

This study was conducted as a case study using data collected from a major IT company, IBM. The research programme comprised of two sections: the statistical analysis of data provided by IBM; and the analysis of interview data collected from IBM practitioners involved with CBD using the Grounded Theory Method.

The objective of the statistical analysis section was to determine if there was evidence to support or refute the claimed cost benefits of CBD. This included testing the CBS Functional Density Rule of Thumb (Abts, 2002; 2004).

The objective of the Grounded Theory section was to conduct a focused investigation of the cost factors affecting CBD and to offer an explanation of the results originating from the statistical analysis section.

3.2 Adoption of a Case Study Approach

IBM is a multinational corporation employing over three hundred thousand people worldwide. Furthermore, IBM was deemed to be of a sufficient size to be involved in all manner of IT development and management activities including the development of COTS-based and custom built systems. Therefore, it was felt that there was the potential to gather data relating to a wide range of global software projects involving different system types.

It was assumed that data received from IBM was standardised. The original belief was that the researcher would be able to discuss the nature of any data with those providing the data because the researcher worked for the company. Conversely, it was believed that if data was provided by other companies confidence of its validity could not be
established; a researcher not working for other companies may not be in a position to question the source of data.

For this research project it was considered beneficial to collect data from only one company, IBM. The reasons for this decision are as follows:

The researcher was employed by the company. It was assumed that this factor would grant the researcher access to data sources which would not be available to other researchers not employed by the company. It was also considered that because the focus of this research programme was on the cost of systems other companies would not be willing to divulge this type of data to those outside of the their organisations because of its commercial sensitivity or confidentiality.

The organisational structure of IBM incorporates an internal global profession’s hierarchy. The following are examples of these professions: IT Architect, IT Specialist and IT Consultant. Furthermore, employees were assessed against a set of skills related to their profession and then associated with a level of competence. The aim of this structure was to objectify the skill level of staff throughout the company. This factor was considered important for this study because it added extra confidence in the control of variables relating to job responsibility and skill level. Thus, staff belonging to the same profession and skill level should be performing comparative tasks and display similar skill profiles. With data collected from other companies it was felt that an IT Architect in one company may perform a totally different role compared with someone with a similar job title who worked for another organisation.

IBM uses a standard set of ‘global’ software development methods for software development projects. These methods are essentially reusable ‘intellectual property’ design guidelines. For example, there are methods for COTS-based system development and other methods for custom projects. It was felt that for the purposes of this study the use of global methods for all project types would add extra control to the variables relating to the software development environment. Thus, the assumption was that any difference in ‘cost’ would be attributed to factors other than the development methodology.
Other studies which investigated the cost factors of COTS-based systems referenced data collected from government, university and other commercial sources. For example, the *CBS Functional Density Rule of Thumb* (Abts, 2002; 2004) was developed with reference to data with a bias towards large aerospace and defence organisations based in the United States. Therefore, it was felt that collecting data from a large corporation from the commercial business sector would add a degree of originality to this study.

However, a limitation of results from studies where data originated from only one company was with the confidence that the results could be generalised outside of that organisation (Kitchenham, Pfleeger, McColl & Eagan, 2002).

### 3.3 Justification for Using the Case Study Research Method

“A case study is an empirical inquiry that investigates contemporary phenomena within its real-life context, especially when the boundaries between phenomenon and context are not clearly evident” (Yin, 2003, p. 13). In this context *phenomenon* was considered to be related to the states or to the events that affect the study focus (Weber, 2003). For this PhD research project, phenomena were identified as entities such as cost, organisational relationships, software maintenance tasks and the software engineering processes used to develop COTS-based software systems.

Inferential statistics were deemed appropriate for the exploratory phase of this study because the data were ratio in nature. Inferential statistics were used to test for significant relationships (correlations) between the data provided by IBM. However, inferential statistical tests cannot be used to determine the cause of any correlations (Robson, 2003).

The aim of the Grounded Theory section of this study was to investigate and understand the cost factors and issues relating to CBD. The proposed emphasis of this research phase was explanatory, focusing upon system development and maintenance tasks and relationships occurring over time, rather than mere frequencies or incidence. The Case Study research method was considered suitable for this type of study where an understanding of causal mechanisms is required (Yin, 1994; Bennett, 1997; Yin, 2003).
This also formed the basis for choosing the Case Study research method for this research project.

It was envisaged that a flexible research design, which the case study method is classified as (Robson, 2003), was more suitable for this research project than a fixed research method; prior to the data collection phase no knowledge existed about the nature of available data sources or characteristics of interview participants. One feature of a fixed research design method is the pre-selection of variables to be used in the study (Robson, 2003). This factor was deemed impractical because these variables were not known at the outset of the study.

Another reason for adopting a flexible research design approach was the assumption that the focus of the investigation would change and evolve in response to the data collection phase. For example, when collecting data using the interview technique it may become clear from the responses of the participants that areas of the research domain not initially regarded as being important should in fact need to be considered. This was in contrast to fixed research design approaches, such as an experiment or multiple choice questionnaires, where variable choices tend be defined early on in the design (Robson, 2003). For this study it was considered that one area of investigation could lead to other areas of interest. Therefore, flexibility of design was considered important to react to new areas of concern. The above reasons provided further justification for selecting the Case Study research method.

Another key criterion for research method selection concerned the nature of the research questions (Dube & Pare; 2003, Yin 2003). The case study research method was deemed appropriate for the investigation of ‘why’ question types because there may be a requirement to deal with operational links needing to be traced over time rather than just frequency of incidence (Yin, 2003). For this study one research question originating from the literature review was a ‘why’ question: ‘Are COTS-based system total lifecycle costs significantly different from custom-built software systems and why?’ This originated from Reifer et al. (2003; 2004), who suggested that as a result of significant maintenance costs the total cost of ownership of COTS-based systems could be higher than equivalent custom built systems. However, another research question, ‘Was there evidence to support, refute or extend the CBS Functional Density Rule of Thumb (Abts, 2002; 2004)?’, related to understanding why consideration of the CBS
It should be noted that there were other research methods which were suitable for dealing with ‘why’ type questions. One example is the experimental research method (Yin, 2003). However, the experimental research method was discarded for this PhD research project due to the constraints of time, finances and other resources. For example, for this study there was a time limitation of the equivalent of three years full-time study. It is assumed that an experimental investigation into the factors affecting the total lifecycle costs of COTS-based systems would need to involve studying both the development and maintenance phases of such systems over a period of time exceeding the allocated research time. Another experimental design option was to design and implement a COTS-based system, recording the lifecycle costs throughout its life. However, the available financial resources were not considered to be sufficient to build a suitable software system for use in this study. Some business systems are developed over many years by teams of software engineers. There were no available funds to employ teams of software developers. However, it was deemed feasible for the researcher to study data from existing software systems. Thus, the use of the Case Study research method was considered a more appropriate approach for this study.

This research project was also deemed to have an explanatory focus. For example, one aim of the study was to assess if there was evidence to support the CBS Functional Density Rule of Thumb (Abts, 2002; 2004) and thus, to explain the relevance of the CBS Functional Density Rule of Thumb within the area of system design. Another research method considered was the Survey research method (“Survey and Questionnaire Design”, 2007). However, Robson (2003) stated that the survey research technique was primarily a descriptive research method. For this reason, the survey research method was considered unsuitable for explanatory research (Robson, 2003; Yin, 2003) and thus was deemed inappropriate for use on this research project.

One reported criticism of the Case Study methodology was the inability to generalize to a wider population from the results of a single case (Yin, 2003). For example, Yin (2003) implied that the results of one case study could only be legitimately applied to that case. However, a similar criticism could also be applied to more established methods, such as, the experimental research method (Robson, 2003). For example, in order to be able to
justifiably generalise from the results of a controlled experiment it could be suggested that the same exact conditions of that experiment must also exist.

One method for addressing the criticism of generalisability is to study multiple cases because this allows evidence from one case to support and to be compared with evidence gained from other cases (Burton, 2000; Yin, 2003). For example, another case could be a different company. However, a multiple case study design, involving the collection of data from additional companies was not selected for this research programme because, as mentioned earlier, it was felt that IBM was such a large company that it constituted a valid case in its own right.

The lack of rigor has been cited as a criticism of the Case Study Method (Yin, 2003). Examples of where a lack of rigor can occur include case studies in which researchers let their biased views influence the direction of the findings and conclusions or where they fail to follow systematic procedures (Yin, 2003).

Another criticism of the Case Study Method is reactivity, which occurs when the presence of a researcher alters the results of a study (Yin, 2003). For example, interview or direct observation data collection methods require the presence of the researcher so could be affected by reactivity.

### 3.4 Data Collection

The following sections discuss the collection of data.

#### 3.4.1 Triangulation

Triangulation is a proposed method to enhance the rigor of a case study (Robson, 2003). Triangulation involves gaining support for any research findings from multiple sources of evidence. An example of triangulation can involve supporting the results of statistical analysis with evidence from a literature review and verification from the analysis of interviews. Thus, with triangulation conclusions can be considered more reliable because findings gained from one source of evidence can be used to verify findings originating from other sources (Robson, 2003). For this study the objective was to employ the concept triangulation. However, it was not possible to link the results of the Statistical Analysis to the practitioners interviewed in the Grounded Theory section. Details of the people involved in the projects represented by the data used in the
Statistical Analysis section were not available. This was an unavoidable limitation of this study.

Triangulation is also considered to reduce the threat of reactivity when sources of evidence such as documentary evidence, the contents of which tends not to be collected by the researcher, is used to support findings derived from other evidence sources which tend to directly involve the researcher, such as interview evidence (Robson, 2003).

For this study, the aim was to collect and analyse documentary data as part of the Grounded Theory section. However, suitable documentary evidence was not available.

3.4.2 Identifying Data Sources

The initial requirement of this study was to identify suitable sources of data. For the Statistical Analysis, which was exploratory in nature, the aim was to identify data sources to be used to investigate the cost factors of COTS-based systems.

The method used to identify information sources was by email enquiry. This was initiated by initially asking senior members of IBM’s technical community to recommend suitable people to contact. These people were then asked by email if they could supply information relating to COTS-based or custom built system lifecycle costs. If they were unable to supply any information they were requested to recommend other suitable contacts. It was believed that this approach added a degree of flexibility to the identification of data sources as respondents were free to suggest any data they considered to be relevant. This approach was also used to identify the practitioners interviewed as part of the Grounded Theory section.

The use of email was deemed to be an appropriate approach to identify sources of information for the following reasons:

• E-mail use in research is suitable for use with samples such as company employees, where e-mail access can be high (Burton, 2000). For this study, it was assumed that all employees of IBM, a major technology company, were contactable by email.
• It is a cost effective method for contacting people. Unlike the postal service, email incurred no cost.

• The speed advantages of email make it highly suitable for the initial identification of suitable sources of information (Burton, 2000). However, it relied on recipients replying to messages. It has been suggested that the more attempts to reach respondents, the higher the response rate (Burton, 2000). Therefore, the ‘return receipt’ option was selected for each email sent. This feature enabled the researcher to receive the delivery confirmation status of each email sent to recipients. It also returned a message confirming when the email was opened by a recipient. Furthermore, the IBM policy is for employees to activate an ‘out of office’ message during periods when they are not working. These email features enabled decisions to be made on whether to target alternative people if it was clear that the original recipients were either away from their workplace or had not opened and read the emails within a reasonable period of time.

However, a challenge of using email to identify information sources and to gather data concerns the control of external validity in a research design (Robson, 2003). External validity should be controlled by including sampling of the focus population (Robson, 2003). However, in this study the aim of the email enquiry was to identify available data sources. Therefore, it was deemed impractical to sample the population.

3.5 Statistical Analysis

The aim of this section is to detail the design of the statistical analysis section of this study.

3.5.1 Requested Data Types

To enable an understanding of the factors which influence the costs of CBD data variables adhering to the following criteria were deemed important. The reasons for these choices are given below:

• A figure relating to the lifecycle costs of systems. It was assumed that the total lifecycle costs of systems was made up of different constituent cost values.
Examples of these were development costs, maintenance costs, infrastructure costs, human effort, consultancy, costs to software component purchase costs etc. However it was deemed important for data relating to cost to contain a clear definition of zero; zero cost should imply no cost. Therefore, cost was defined as a ratio variable (Brace, Kemp & Snelgar, 2006).

- The percentage of system functionality supplied from COTS components. This was defined as a ratio variable because zero COTS functionality implies a system containing no COTS-components. This was considered to be a valid variable because COTS-based systems may not be built entirely from COTS components. For example, COTS-based systems were defined as systems built from over thirty per cent COTS components (Boehm, Port, Yang & Bhuta, 2003). Thus, the percentage of functionality supplied from COTS components may vary for different systems. Furthermore, this variable was required to assess the validity of the CBS Functional Density Rule of Thumb (Abts, 2002; 2004) because a constituent part of this rule of thumb is the amount of system functionality supplied from COTS components.

- The number of COTS components used in each system and their functional size. It was considered that these variables were integral constituents of the CBS Functional Density Rule of Thumb (Abts, 2002; 2004), which states that to minimise maintenance costs systems should be built from the least number of COTS components. However, it should be noted that data adhering to this requirement was not provided during the collection of data.

- The size of each system. To compare system costs it was judged important to understand the relative size of each system because system size was claimed to be a reliable predictor of system development effort. Thus, the perception was that larger systems would be associated with higher costs and visa versa (Abrahão & Pastor, 2003). Measures of system size are function points or source lines of code (Ahmed, Bouktif, Serhani & Khalil, 2008; Zheng, Zeng, Wang & Shi, 2009).
• System complexity. The quantification of the complexity of functions being integrated into each system was deemed to be a contributing factor of cost (Minkiewicz, 2004). The assumption was that software functions performing tasks, such as a simple numeric addition or subtraction were considered less complex and thus easier to modify, integrate and test than software performing intricate real-time processing (Minkiewicz, 2004). However, details on system complexity were not made available as part of the data collection for the statistical analysis section.

3.5.2 Data Analysis Design

An understanding of how data is to be analysed should be considered during the design stage of a research project (Robson, 2003). Thus, with this in mind, quantitative data was collected and analysed using a combination of descriptive statistics and inferential statistics. The reasons for these choices are as follows:

Descriptive statistics should always be included in a research report to summarise and provide information about the data sample before performing inferential statistics (Brace et al., 2006). Thus, scatter diagrams, histograms and tables displaying the arithmetic mean, median, mode, standard deviation, variance, skewness and kurtosis were descriptive statistical tools used to provide a visual display and summarise the data. Another purpose for presenting the descriptive statistics was to determine the data’s suitability for parametric tests. For example, the assumptions are that data are normally distributed, should be measured at least at the interval scale and that the data from different systems are independent (Field, 2009). The assumptions of interval and independent data can only be tested by commonsense. However, normal distribution can be assessed objectively with tests (Field, 2009). The Kolmogorov-Smirnov test of normality is considered useful to indicate that data approximates a normal distribution (Coakes, Steed & Ong, 2009). Further details of the descriptive statistics used in this study can be found in Chapter 4 and Appendix A.

Inferential statistics are procedures which go beyond describing data and were used to answer questions about the data. Inferential statistics differ from descriptive statistics in that mathematical procedures are used to estimate the probability that the data support a hypothesis (Brace et al., 2006). Details of the inferential tests used can be found in section 3.5.4.
The data was analysed using PASW (Predictive Analysis SoftWare) version 18 [Note: SPSS software after version 17.0.2 was renamed PASW]. PASW was chosen because it is recognised as a leading quantitative data analysis tool (Kerr, 2002; Brace et al., 2006; Coolican, 2009; Field, 2009) and recommended by the University of Portsmouth. However, alternative data analysis tools were available, such as Microsoft Excel. However, the data analysis component of Microsoft Excel offered fewer parametric and non-parametric tests, compared with those included with PASW.

### 3.5.3 Hypotheses Tested

The hypotheses which were tested are detailed in Section 4.4. It should be noted that hypothesis selection was made with reference to the data and available variables. Details on the nature of the data which was collected from IBM can be found in Chapter 4. Furthermore, Chapter 4 also provides justification for the choice of each hypothesis.

The commonly accepted criterion for the use of inferential tests is to define probability (p) at 0.05 (Coolican, 2004; 2009). The interpretation of this figure indicates that if the probability value is calculated to be equal or less than 0.05 then the null hypothesis can be rejected and the result defined as being statistically significant (Chalmers, 1999; Coolican, 2004; Brace et al., 2006; Coolican, 2009).

### 3.5.4 Inferential Statistical Tests

The choices of inferential statistical tests were dependent upon the nature of the data. The following tests were selected with reference to the data following its collection:

1) **Correlation.** Correlation is the measurement of the extent to which pairs of related values on two variables tend to change together (Coolican, 2009). Thus, a positive correlation occurs when the value of one variable increases with the other. Conversely, a negative correlation occurs when the inverse is true. The level of correlation is measured by the correlation coefficient; the closer the value is to ‘1’ (or ‘-1’) the stronger the relationship. A lack of correlation is suggested by a value close to zero.

A test of correlation was selected to test the degree of association between pairs of variables. Details of the variables which were collected can be found in Section 4.3.
Furthermore, a test of correlation was deemed suitable because the variables were collected simultaneously and not manipulated, as would be the case with an experiment (Coolican, 2009).

Coolican (2009) stated that there are two main tests of correlation: Pearson’s Product-Moment Correlation Coefficient, which is a parametric test, and Spearman’s Rank Correlation, which is the non-parametric test. However, the raw data did not adhere to the one of the requirements for Pearson’s Product-Moment test of correlation; the data should be normally distributed (Brace et al., 2006; Coolican, 2009; Field, 2009). To address this, a logarithmic transformation of the data was performed (Log Base 10). Pearson’s Product-Moment test of correlation was used to test the association between pairs of the variables. The results of this analysis and justification for the logarithmic transformation of the data can be found in Sections 4.7.1 and 4.6 respectively.

2) Multiple linear regression. This involves the statistical prediction of one variable from the correlations of other known variables (Coolican, 2009). The use of multiple linear regression models were chosen to explore the nature of the relationships between more than two variables. Further details of these tests can be found in Chapter 4.

It should be noted that an original aim was to compare the lifecycle costs of COTS-based and custom-built systems, for which the Independent T-Test or Mann-Whitney U Test were deemed appropriate. These tests are suitable for the comparison of two variables (Coolican, 2009). However, it was not possible to justify the grouping of the data into two independent variables which represented COTS-based or custom-built systems. Thus, for this reason the use of these tests were rejected for this study.

3.6 Grounded Theory

The aim of the Grounded Theory analysis was to conduct a focused investigation of the factors influencing the costs of CBD, using data collected from IBM practitioners who had been involved with building systems from COTS-based components, to offer an explanation of the results originating from the Statistical Analysis section. The following sections provide details of the Grounded Theory method and justifications for its choice in this case study.
3.6.1 Data Collection Methods

There are different sources of evidence which are deemed suitable for use with case studies (Robson, 2003). These range from documentation and archival record searches to interviews and direct observation. (Yin, 2003) However, although no single source of evidence is considered to have a complete advantage over the others the interview technique was judged to be an essential source of case study information (Yin, 2003).

The interview data collection technique was proposed as the primary data collection method. The reason for this choice is given below.

3.6.1.1 Interview Data Collection Method

An advantage of the interview data collection technique is that it can enable an interviewee to organise and describe their view of the world, emphasising the issues they find important (Kvale, 1983). Thus, participants can be asked about past events, their views on future events and what they are doing at the present time. Further benefits of the interview data collection method are that that the questioning technique, focus and approach can be flexibly adjusted by the interviewer during the interview (Yin, 2003). A further reason for choosing the interview data collection technique was that it was considered feasible to perform. There were CBD practitioners within IBM who were willing to be interviewed. These points were deemed important in this study in order to exploit the benefits of a flexible research design.

The above points are in contrast to alternative data collection methods, such as the multiple-choice questionnaire or survey designs, where the questions and answers are previously proposed by the researcher (Kvale, 1983). Thus, it was felt that these types of research methods were more suitable for a fixed research design, not a flexible research design, which was the proposal for this part of the study.

Furthermore, it was felt that a limitation of the questionnaire/survey approach to data collection was that once the set of questions had been designed and submitted to the subjects it would be difficult to gather additional information not covered in the original set of questions (Robson, 2003). Additionally, it was considered that not enough was known, by the researcher, about issues affecting CBD for adequate questions to be pre-designed. Thus, the questionnaire and survey data collection methods were deemed
inappropriate because the aim was to gain a better understanding of CBD cost factors and not to just confirm preconceived ideas.

3.6.1.2 Direct Observation

Direct observation (Robson, 2003) was another data collection method which was considered. However, this method was only deemed suitable if participants are involved in activities relating to the area of study, such as, meetings, workshops, customer briefings during the time set aside for data collection. However, there were no opportunities to perform direct observation. The benefit of the interview data collection method is that it is not dependent on specific activities occurring at certain times.

3.6.1.3 Document Search

It should be noted that the collection of documentary evidence was initially proposed as a secondary data collection method. Sources of documentary evidence included design documents, meeting minutes, strategy documents and internal emails and reports (Walsham, 1994). However, relevant sources of documentation were not available.

3.6.2 Data analysis Approach

The analysis of case study evidence was cited as being one of the least developed and most difficult aspects of conducting case studies (Yin, 2003). Furthermore, there was a lack of common methodological principles for the analysis of data collected by the interview technique (Kvale, 1983) and that careful consideration of the choice of data analysis methods were required to avoid researcher bias (Robson, 2003).

A recommended approach was for the partial analysis of data early on in a case study, rather than to wait until all of the data had been collected before beginning the analysis (Miles & Huberman, 1994). An advantage of adopting this type of iterative approach to data analysis was that it allowed the researcher to modify the data collection focus in reaction to any interesting results. This approach was also considered to embody the ethos of a flexible research design which was regarded as an important facet of this part of the study.

There were other potential options regarding the analysis of data. An important factor relating to the choice of methods concerned the nature of the data. The assumption was that data resulting from an interview would be primarily text based. For example, this related to field notes created during or following an interview. Thus, it was important to
select a data analysis method which was suitable for the analysis of this type of data. Furthermore, there was the potential that the interview data collection method would yield a great deal of material. It was therefore also deemed important to consider a data analysis method which aided the reduction and organisation of the data into manageable parts (Robson, 2003).

Miles and Huberman (1994) explained that with text based data it is not the words that matter but their meaning. One method to attribute meaning to this type of data is coding (Robson, 2003). Codes are classed as tags or labels used to organise and assign units of meaning to the data. Codes can also be associated with varying sizes of data chunks, such as, words, sentences, paragraphs etc. Coding systems range from categorisation labels to complex metaphor labels (Miles & Huberman, 1994).

A coding method preferred by Miles and Huberman (1994) was the creation of a provisional ‘start list’ of codes prior to beginning fieldwork. However, this type of coding scheme was deemed unsuitable for this study because it relied on commencing data analysis with a set of preconceived codes, which it was felt would contribute to forcing the outcome of the analysis.


In relation to this study, the Grounded Theory method was selected for the following reasons:

- The Grounded Theory coding process allows large amounts of data to be analysed quickly (Glaser, 1978).
- All data sources can be analysed as part of the Grounded Theory process. Thus, for example, the Grounded Theory method can be used to analyse data originating from interviews and documents. However, although the intention was to collect documentary data this type of data was not ultimately made available by IBM.
- The Grounded Theory method claims to remove personal bias from the analysis of data (Glaser, 1978; Charmaz, 2006) by assigning concepts to the data. Glaser
(2001) stated that one important property of conceptualisation was that concepts should be abstract of time, place and people. Glaser (2001) further claimed that conceptualisation assisted with divorcing the analysis from personal bias and perceptions. However, it should be noted that it may not be possible to completely remove personal bias from the analysis of data because it is not possible to prevent researchers from referencing past experiences.

- The weakness of some other data analysis methods was that they could only be used to confirm preconceived ideas. Grounded Theory is proposed as a valid method for identifying new ideas (Glaser, 1978; Charmaz, 2006). This was deemed to be an attractive feature of the Grounded Theory method because the aim of this Grounded Theory analysis was to develop a new understanding of the factors affecting CBD costs and which explained the results of the Statistical Analysis.

Details of the Grounded Theory method are presented below.

### 3.6.3 Grounded Theory Data Analysis Method

The Grounded Theory methodology was originally developed by two sociologists, Barney Glaser and Anselm Strauss in order to deal with research situations not suited for the collection of quantitative data to be analysed by statistical methods. The term ‘grounded’ was used because the aim of the method was to develop theories which were grounded in the data. The premise was that theory derived from data was more likely to resemble ‘reality’ than theory derived from the interpretation of empirical evidence (Glaser, 1978; Strauss & Corbin, 1998; Locke, 2005; Charmaz, 2006).

There are different versions of the Grounded Theory method. However, the most popular choices are Glaser (1978), Strauss and Corbin (1998) and Charmaz (2006). The Glaser and Strauss alliance gradually separated until each was developing different versions of the methodology (Coleman & Connor, 2007). Strauss and Corbin (1998) created an updated version of the Grounded Theory method with an extended coding system. However, this drew criticism from Glaser (1992) for being formulaic and thus forcing a theory from the data. Strauss & Corbin (1998) rejected this criticism stating that their method did not force theories but allowed the ‘data to speak.’ Other differences between the Grounded Theory flavours are as follows:
• Glaser (1978) proposed the conceptualisation and generation of theoretical explanations of substantive areas;
• Conceptual description of the study area was the focus of Strauss and Corbin’s (1998) Grounded Theory method.
• The focus of Charmaz’s (2006) Grounded Theory flavour was more on themes which occurred within the data, rather than concepts and categories.

Glaser (1978; 1992; 1998) and Strauss and Corbin (1998) also differed on other fundamentals parts of the Grounded Theory method. Glaser believed that a Grounded Theory study should not begin with a preconceived theory in mind. The research problem and question should be allowed to emerge from the data. Thus, the premise was that knowledge development begins with knowledge generation rather than knowledge verification. The focus was on conceptualisation and generating a theoretical explanation of a substantive area (Glaser, 1978; 1992; 1998; Coleman & Connor, 2007). Conversely, Strauss and Corbin (1998) recommend defining the research question beforehand. They suggested that this helped define the scope of a study. Further differences between Glaser’s and Strauss and Corbin’s Grounded Theory methods concerned the place of the literature review. Glaser (1978; 1992; 1998) stated that a literature search should not be conducted before the start of a study to avoid creating preconceptions about the area of study. However, Strauss and Corbin (1998) recognised the importance of understanding the literature before undertaking a study and referring to it when required.

On data collection Glaser’s (1978) Grounded Theory approach differed from other Grounded Theory methods. Glaser (1978) advised against recording interviews to avoid the generation of too much data. He suggested that field notes should be created from interviews to capture the main points; the process of creating field notes was viewed as an early level of coding in which the researcher only recorded details deemed important. Strauss and Corbin (1998) and Charmaz (2006), on the other hand, supported the recording of interviews. Their justification was that this allowed for interview data to be revisited many times.
However, a consensus of the different Grounded Theory methods concerned when data collection and analysis should occur. Glaser’s (1978; 1992; 1998), Strauss and Corbin’s (1998) and Charmaz’s (2006) Grounded Theory approaches prescribed that data collection and analysis should occur at the same time. For example, the analysis of the first interview should dictate the focus of the second interview and further interviews (Coleman & Connor, 2007). This feature supported the notion of a flexible research design which was proposed as a feature for this study, thus providing further justification for the use of the Grounded Theory method in this study.

The aim was to base the Grounded Theory analysis on Glaser’s (1978) Grounded Theory method. This method was chosen because Glaser (1978; 2001) claimed that as a result of the processes of conceptualisation and constant comparison, his method was more rigorous than the other Grounded Theory flavours. Furthermore, Glaser (1978) proposed that as a result of the rigor of his method that a new understanding or theory would naturally emerge from the analysis. However, it should be noted that Glaser’s (1978) method could not be adhered to completely because by the time the decision to adopt the Grounded Theory approach to data analysis had been made the literature review had already been performed.

The following sections provide details of the stages involved with the Grounded Theory method.

**3.6.3.1 Open and Selective Coding**

Coding is the part of the analysis concerned with identifying, naming and categorising phenomena and incidents found in the text. The first part of this process was termed ‘open coding’ because the task of attributing meaning to data should be performed in any way which seemed relevant and should be performed with an open mind (Glaser, 1978; Allan, 2003). When coding the following questions should be asked of the data: What is this data a study of? What is actually happening in the data? What are the main concerns being faced by the participants? What accounts for the continual resolving of this concern? (Glaser, 1978). The aim of these questions is to focus the mind on attributing patterns among incidents that yield codes at the conceptual level, as opposed to choosing purely descriptive codes (Glaser, 2001).
The coding process focus should change to selective coding as categories and a central theme to the theory emerge from the data (Glaser, 1978). Selective coding served to direct the analysis in order to further develop the central theme and to avoid wasting time continuously analysing all areas of concern (Glaser, 1978). Selective coding also allowed further data to be coded quickly.

### 3.6.3.2 Constant Comparison

Constant Comparison was considered to be an integral part of the Grounded Theory method because this was claimed to enable theories to emerge from the data (Glaser, 1978; Allan, 2003; Locke, 2005). The constant comparison process should be performed in conjunction with open coding and selective coding and involves constantly looking for patterns, similarities and differences within the data. The first part of the constant comparison process involves comparing each open code with every other open code to identify concepts which explain the underlying processes (Glaser, 1978). This process serves to compare incident with incident, encompassed within the open codes, to explain what is going on within the data. As concepts emerge the constant comparative process should focus upon comparing concepts to more incidents to generate theoretical properties of the concepts. This enables the concepts and conceptual properties to emerge. In conjunction, concepts should be compared with other concepts to establish categories and links to the emerging theory (Glaser, 1978). Thus, the constant comparative process is applied at different levels of granularity; it is used to compare data with data, codes with codes, codes with concepts, concepts with concepts, concepts with categories and categories with categories.

Concepts and categories were defined as empirical abstractions (Skemp, 1986; Allan, 2003; Locke, 2005). Glaser (1978; 2001) and Strauss and Corbin (1998) disagreed on the degree of abstraction involved with conceptualisation. Glaser (1978; 2001) prescribed that conceptualisation should be performed independently of time, place and people. His justification of this was that concepts generated independently of context could be generalised to other areas of research. Strauss and Corbin (1998), however, recommended that the conceptual name should be associated to the context of the data in order to add meaning to the concepts. However, Glaser (1978; 2001) dismissed Strauss and Corbin’s (1998) approach to conceptualisation because he felt that it served to force descriptions on concepts with reference to their context, reducing their abstraction and thus, their ability to be generalised.
3.6.3.3 Memoing

The writing of memos facilitated the constant comparison process. A memo was defined as a free-format piece of written work which can vary from a few lines to a few pages in which links between open codes, concepts and categories could be discussed (Allan, 2006). The main aim of memoing is to reach clarity of thought with a freedom to investigate ideas (Allan, 2006).

Glaser (1978) defined several rules of writing memos. For example, one recommendation was that data should not be put in memos, with the exception of clearly demarcated, useful illustrations referenced to the field notes from where they came. However, Charmaz (2006) argued that anything can be placed in memos as the purpose of a memo is to develop critical and analytical thinking, forming the basis of inference and interpretation of the links between patterns in data.

3.6.3.4 Theoretical Sampling

Theoretical Sampling is a technique used to facilitate the identification and definition of categories and their properties. This involved identifying and collecting relevant data, specifically to elaborate and refine a category. The aim was to continue this process until no new properties emerged. When this condition was reached Glaser (1978) referred to this as saturation.

3.6.3.5 Theoretical Coding

Later stages of Glaser’s (1978) Grounded Theory method involved grouping and regrouping codes into lists of broader concepts and then into even broader categories. It was the investigation of the connections and relationships between concepts and categories and categories with other categories, which Glaser (1978) claimed, would allow a theory to emerge. This process was referred to as Theoretical Coding (Glaser, 1978; 2001).

However, it should be noted that the process of Theoretical Coding differs between the Grounded Theory flavours. For example, Glaser (1978; 2001) and Strauss and Corbin (1998) disagreed on how and when it should be performed. Strauss and Corbin’s (1998) approach was to commence Theoretical Coding early on during the analysis, even when performing open coding. They termed this as Axial Coding because they deemed this analysis process to occur around the axis of a category. A criticism of Strauss and Corbin’s (1998) approach to Theoretical Coding was that performing Axial Coding,
whilst also carrying out *open coding*, could take the focus away from finding the real issues within the data (Allan, 2006).

Glaser (1978) prescribed performing *Theoretical Coding* later on in a Grounded Theory study. His argument was that during the open coding phase not enough was known about the area of study to start establishing theoretical links. Thus, performing Theoretical Coding following the identification of concepts and categories enabled a fuller picture of the study area and a better understanding of how the Theoretical Codes should fit.

Glaser’s (1978, 1998) Theoretical Coding process involves linking one or more of eighteen coding families to the *concepts* and *categories* which emerged from *open coding*. These codes are presented in Table 3.1. Glaser (1978) stated that a theory would emerge as a result of performing Theoretical Coding and creating memos to articulate the associations and disparities between combinations of categories and concepts.

<table>
<thead>
<tr>
<th>Coding Families</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Six C's</td>
<td>Causes (sources, reasons, explanations, accountings or anticipated consequences), Context or Ambiance, Contingencies, Consequences (outcomes, efforts, functions, predictions, anticipated/unanticipated), Covariances, Conditions or Qualifiers.</td>
</tr>
<tr>
<td>Degree</td>
<td>Limit, Range, Intensity, Extent, Amount, Polarity, Extreme, Boundary, Rank, Grades, Continuum, Probability, Possibility, Level, Cutting Points, Critical Juncture, Statistical Average (mean, medium, mode), Deviation, Exemplar, Modicum, Full, Partial, Almost, Half.</td>
</tr>
<tr>
<td>Dimension</td>
<td>Dimensions, Elements, Divisions, Piece of, Properties of, Facet, Slice, Sector, Portion, Segment, Part, Aspect, Section.</td>
</tr>
<tr>
<td>Type</td>
<td>Type, Form, Kinds, Styles, Classes, Genre.</td>
</tr>
<tr>
<td>Interactive</td>
<td>Mutual Effects, Reciprocity, Mutual Trajectory, Mutual Dependency, Interdependence, Interaction of effects, Covariance, Face to Face Interactions, Self-indications,</td>
</tr>
<tr>
<td>Coding Families</td>
<td>Examples</td>
</tr>
<tr>
<td>----------------</td>
<td>----------</td>
</tr>
<tr>
<td>Cutting Point</td>
<td>Boundary, Critical juncture, Cutting point, Turning point, Benchmark, Division, Cleavage, Scales, In-out, Intra-extra, Tolerance levels, Dichotomy, Trichotomy, Polychotomy, Deviance, Point of no return.</td>
</tr>
<tr>
<td>Means-goal</td>
<td>End, Purpose, Goal, Anticipated consequences, Products.</td>
</tr>
<tr>
<td>Cultural</td>
<td>Social norms, Social values, Social belief, Social Sentiments.</td>
</tr>
<tr>
<td>Consensus</td>
<td>Clusters, Agreements, Contracts, Definitions of Situation, Uniformities, Opinions, Conflict, Discensus, Differential perception, Cooperation, Homogeneity-heterogeneity, Conformity, Non conformity, Mutual expectation.</td>
</tr>
<tr>
<td>Mainline</td>
<td>Social control, Recruitment, Socialization, Stratification, Status passage, Social organization, Social order, Social interaction, Social mobility.</td>
</tr>
<tr>
<td>Theoretical</td>
<td>Parsimony, Scope, Integration, Density, Conceptual level, Relationship to data, Relationship to other theory, Clarity, Fit, Relevance, Modifiability, Utility, Condensability, Inductive-Deductive balance and interfeeding, degree of, Multivariate structure, Use of theoretical codes, Interpretive, Explanatory, Predictive Power.</td>
</tr>
<tr>
<td>Ordering or Elaboration</td>
<td>Structural Ordering (unit size of: organization, division...), Temporal Ordering (A--&gt;B--&gt;C), Conceptual Ordering (Achievement Orientation, Institutional Goal, Organizational value, Personal Motivation).</td>
</tr>
<tr>
<td>Reading</td>
<td>Concepts, Problems, Hypotheses.</td>
</tr>
<tr>
<td>Models</td>
<td>Linear model, Property Space</td>
</tr>
</tbody>
</table>

Table 3.1: Glaser’s eighteen theoretical coding families (Glaser, 1978, pp. 72-82).

With reference to Table 3.1, a criticism Glaser’s (1978) view on Theoretical Coding was that the application of his theoretical codes could be seen to result in forcing preconceived links (Charmaz, 2006), which was in conflict with Glaser’s (1978; 1998) assertion that the aim of his Grounded Theory method was to move away from the preconception of ideas. However, Glaser (1998) defended his position, stating that if his coding families did not fit an analysis then analysts should develop additional coding families or to add extra explanations to the existing coding families. Glaser (1978; 1998) further justified his theoretical coding families, stating that their use encouraged
analysts to maintain a conceptual level in writing about the relationships between concepts and categories.

3.7 Conclusions

This chapter has presented an outline of the research methods employed in this study. The basis of the design was to conduct a case study, collecting data from IBM. A justification of the collection of data from one company was presented, as were details on the identification of data sources and the collection of data.

The research design comprised of an exploratory section in which the statistical analysis of data relating to CBD lifecycle costs was performed. The results of the Statistical Analysis can be found in Chapter 4.

The design of the second section involved the use the Grounded Theory method to analyse interview data provided by IBM practitioners involved in CBD. The aim of this section was to gain a greater understanding of the results of the Statistical Analysis and to gain a better understanding of the factors affecting the costs of CBD. The results of the Grounded Theory analysis are presented in Chapter 5.
Chapter 4 Statistical Analysis

4.1 Introduction

This chapter presents the statistical analysis of the data provided by IBM, which is discussed below.

The aim was to investigate whether there was evidence to support or challenge the perceived benefits of CBD using data supplied by IBM. The claimed benefits of building systems from COTS components, compared with custom system development, were lower development time and costs and reduced maintenance and evolution costs (Heineman & Councill, 2001; Chung & Cooper, 2002; Seacord & Wrage, 2002; Albert & Brownsword, 2002; Clemente & Hernandez, 2003; Kotonya et al., 2003; Bhuta et al., 2005; Li et al., 2006; Lin et al., 2007; Boehm & Bhuta, 2008; Finnegan, 2008; Sheng & Wang, 2008; Suleiman, 2008).

A further aim of this analysis was to investigate whether there was evidence to support, refute or extend the CBS Functional Density Rule of Thumb (Abts, 2002; 2004).

The CBS Functional Density Rule of Thumb (Abts, 2002; 2004) proposes that in order to minimise development costs the best approach is to maximise the amount of functionality in a system provided by COTS components. However, to minimise system maintenance costs a system should be built from as few COTS components as possible.

Further details on the CBS Functional Density Rule of Thumb (Abts, 2002; 2004) can be found in Section 1.2.

The following statistical techniques and tests were used to analyse the data:

- Descriptive statistics;
- Correlation;
- Multiple linear regression.

The justification for the selection of the statistical tests has been provided in Section 3.5.4.
The data was analysed using PASW (Predictive Analysis SoftWare) version 18.

4.2 Identification of Data Sources and Data Collection Method

The method used to identify information sources was the use of an email enquiry. This was initiated by asking senior members of IBM’s technical community to recommend people to contact who had been involved with CBD. These people were then asked by email if they could supply any information relating to COTS-based or custom built system lifecycle costs. If they were unable to supply this information they were requested to recommend other people.

The result of this approach identified a primary source of evidence, a database which included some cost related data relating to a variety of different projects. Details of the contents of this database are provided below.

4.3 Nature of the Data

The primary source of data originated from a data archive database containing data records for over two thousand Management Information System (MIS) software development projects. MIS is defined as a general term for computer systems in an enterprise that processed information about business operations (Allan, 2005). The term ‘system’ is used to represent either a software system or a software application. For the purpose of data extraction the database was analysed by the database owner (the researcher was not provided with access to the database).

Records for one hundred and fifty eight software development projects were identified as being valid for this study as they contained data fields representing cost (recorded as effort measured in man hours) and COTS-components.

The available variables from IBM’s database were:

- Development effort (System development effort measured in man hours);
- Percentage of COTS components (Percentage of system functionality provided from COTS components);
- System size (System size measured in Function Points);
• **Maintenance effort** (System maintenance effort over 5 year period measured in man hours);

• **Total effort** (Total system life effort (Development effort plus Maintenance effort) measured in man hours).

Reasons for their relevance are presented below:

**Development effort** (System development effort measured in man hours): corresponded to the total number of ‘man hours’ incurred to develop each system up to the point of deployment. However, details on what constituted ‘deployment’ were not available.

System development effort measured in man hours was a measurement of the total amount of time required to develop each system rather than a measurement of elapsed time. However, details of the specific tasks performed or skill levels and number of people involved during system development were not available.

Abts (2004) stated that *effort* was usually substituted as a proxy for cost for the following reasons:

• Financial data was usually not disclosed by companies to researchers; even with a guarantee of a non-disclosure agreement between company and researcher businesses were generally unwilling to release actual cost information to researchers;

• Effort was considered to be a more fundamental quantity for interpretation which could be more readily normalised across diverse organisations. The measurement of effort also did not need to take into account the time value of money – though periodic reassessments need to be performed as technological innovations over time tend to improve productivity.

It can also be seen that in environments where software development tasks are off-shored to countries where financial costs are lower (for items such as salaries) that the measurement of effort is more representative of the tasks performed than financial costs which may differ between countries.
It was also considered that there was a relative association between the number of man hours and financial cost. This association was related to the combined salary, pension and other employment costs of the programmers, system analysts, testers, architects and other people working on each project. Additionally, there were the infrastructure costs of the organisation, such as, the provision of office space, heating, lighting and other services. Thus, the financial costs incurred by a team based in India or China to develop or maintain a system should be lower than if the team was based in Europe or North America where salary and infrastructure monetary costs were higher. However, the tasks should take a comparable amount of man-hours to complete.

**Percentage of COTS components (Percentage of system functionality provided from COTS components):** represented the percentage of COTS components used to build each system. Boehm et al. (2003) defined systems with a percentage of system functionality provided from COTS components in excess of thirty percent as COTS-based systems. However, there was no evidence that IBM applied this definition to categorise systems. Furthermore, details of the development methods, programming languages or hardware requirements for each system were not available.

**System size (System size measured in Function Points):** provided a measurement of the relative size of each system. This measurement was in Function Points. The International Function Point Users Group (IFPUG) standard for measuring Function Points was used. This standard is an implementation of Allan Albrecht’s (Albrecht, 1979) Function Point Analysis (FPA). One of the most important problems faced by software developers concerned how to predict the size of a system and its development effort. FPA was considered suitable for predicting development effort because Function Points could be easily estimated from a statement of basic program requirements early on in the development cycle, unlike Source Lines of Code (SLOC) (Albrecht, 1979; Albrecht & Gaffney, 1983).

FPA measures two components of system size. Firstly, Information Processing Size, which involves counting the number of the following system function components: External inputs and outputs, enquires, interfaces and files. Secondly, Technical Complexity Factor, which consists of fourteen system characteristics which could
influence a system. These range from Communications Facilities through to Multiple Site (International Function Points Users Group, 2004; Allan, 2005).

Software size, measured in Function Points, was also considered to be a reliable predictor of the amount of effort and duration required to maintain systems (Abrahão & Pastor, 2003; Chen, Boehm, Madachy & Valerdi, 2004; Robiolo, Badano & Orosco, 2009). Size and complexity are useful attributes in comparing software systems and can be measured via function-oriented software metrics, such as Function Points (Boloix & Robillard, 1995).

However, a limitation of using Function Points as the only measure of system size and complexity was that this does not consider the work undertaken by system developers and maintainers involved throughout the system life cycle (Jorgensen, 1995; 2004). However, in this study, Function Points were the only available value to use for system comparative purposes.

There are drawbacks with the existing software sizing methods. The following limitations of FPA were proposed (Dolado, 2000; Chen et al., 2004):

- Function Points can only be manually counted;
- There is a high degree of subjectivity in the counting method;
- FPA can be time consuming and expensive;
- The estimator needs to have sufficient experience and expertise with FPA.

The following issues with SLOC were suggested:

- Different language technologies may implement similar functionalities with varying lines of code so this measure can be erroneous.
- SLOC can only accurately be determined once the software system has been built, which is not useful for estimating the size of proposed systems (Dolado, 2000; Chen et al., 2004).

However, for this study SLOC figures were not available from the data supplied by IBM.
**Maintenance effort** (System maintenance effort over 5 year period measured in man hours): was the total number of man hours required to maintain each system over a five year period from 2001 through to 2005. However, it was not confirmed if the maintenance effort measurement covered the whole of 2001 for all systems in the data set. However, it was assumed that the figure represented the whole of 2005 because the data was supplied in 2006.

A breakdown of the tasks performed during the maintenance period was not available. Furthermore, details on what constituted maintenance, as differentiated from system development were not available. Furthermore, it was not known if the systems were to remain in service or be decommissioned after 2005.

**Total effort** (Total system life effort (Development effort plus Maintenance effort) measured in man hours): was the sum of Development effort and Maintenance effort for each system. Therefore, this figure represented the total number of hours of effort required to develop and maintain each system over a period of five years.

### 4.4 Statistical Analysis Plan

The following hypotheses were investigated:

**Hypothesis 1**: System size and Percentage of COTS components are significant predictors of Development effort.

**Hypothesis 2**: System size and Percentage of COTS components are significant predictors of Maintenance effort.

**Hypothesis 3**: System size and Percentage of COTS components are significant predictors of Total effort.

Multiple linear regression models were used to investigate these hypotheses (see Section 4.8) and to explore the nature of the relationships of Development effort, Maintenance effort, Total effort with Percentage of COTS components and System size.
Furthermore, a parametric bivariate correlation (Pearson’s Product Moment Correlation) was used to test the association between pairs of the variables (see Section 4.7.1).

**4.5 Justification for Choice of Hypotheses**

The purpose of this section is to provide a justification for the choice of hypotheses.

**4.5.1 Hypothesis 1**

This hypothesis was selected because the **CBS Functional Density Rule of Thumb** (Abts, 2002; 2004) proposes that in order to minimise development costs the amount of system functionality provided by COTS components should be maximised. Thus, the **CBS Functional Density Rule of Thumb** (Abts, 2002; 2004) suggests a reduction in **Development effort**, in line with an increase in the percentage of system functionality provided from COTS components. However, Abts (2002; 2004) did not explain if a linear relationship between cost and the functionality of a system provided by COTS components was expected.

Furthermore, Hypothesis 1 was chosen to test the effect of building systems from varying percentages of COTS components. Thus, if the premise that CBD should result in reduced development costs was true it could be assumed that systems built from greater percentages of COTS components would incur less effort to develop. One reason for this suggestion was that the greater proportion of development effort should be incurred by the component vendors in developing and testing the components; allowing development cost savings to be passed onto component customers (Couts & Gerdes, 2010).

Another assumption underpinning this hypothesis was that **Development effort** would be higher for larger systems, measured in Function Points. It was deemed valid to also consider the size of a system, as well as the percentage of functionality provided from COTS components, because the measurement of function points have been used for the estimation of effort in different effort prediction models (Albrecht & Gaffney, 1983; Robiolo et al., 2009).
Although it may have seemed obvious that larger systems should incur more development effort, the questions were on the use of FPA (using the IPPUG standard) as a valid predictor of effort and their applicability for the prediction of effort for COTS-based systems. Abts (2004) stated that although FPA was considered an objective means for quantifying the functionality provided by software systems and an accepted alternative to the SLOC metric he suggested that in terms of the measurement of effort, Function Points were calibrated on the amount of time required to deliver code from scratch and that further research was required to determine if there was parity between functionality provided from COTS components and that delivered from custom code, with reference to Function Points measured with the IFPUG method.

Therefore, the aim of Hypothesis 1 was to explore the nature of the relationships between Development effort, System size and Percentage of COTS components.

The null hypothesis is: System size and Percentage of COTS components are not significant predictors of Development effort.

A multiple linear regression model was used to test Hypothesis 1. Coolican, (2009) suggested regression (rather than correlation) should be used when an aim is to predict the value of one variable, with reference to one or more ‘predictor’ variables.

4.5.2 Hypothesis 2

The purpose of Hypothesis 2 was to assess the contribution of Percentage of COTS components and System size as predictors of Maintenance effort. For example, Boehm and Bhuta (2008) suggested that CBD should result in reduced maintenance costs. Therefore, it seemed relevant to investigate whether Maintenance effort was affected by the percentage of system functionality provided from COTS components, with reference to System size, as data representing five years of system maintenance effort was provided by IBM. Thus, with reference to System size it would be expected that Maintenance effort would be reduced with increasing percentages of system functionality provided from COTS components.

Abts (2002; 2004) proposed that to minimise system maintenance costs systems should be built from as few COTS components as possible. However, it should be noted that
details of the number of COTS components making up each system or the proposed life of the systems were not available.

The null hypothesis was: System size and Percentage of COTS components are not significant predictors of Maintenance effort.

### 4.5.3 Hypothesis 3

The purpose of Hypothesis 3 was to assess the contribution of Percentage of COTS components and System size as predictors of Total effort. However, as mentioned earlier, it was not possible to determine how long the systems were planned to remain in service for as only five year’s worth of maintenance effort figures were available. This test was performed to ensure that other relationships which existed between the data were not overlooked.

The null hypothesis for Hypothesis 3 is: System size and Percentage of COTS components are not significant predictors of Total effort.

### 4.6 Preliminary Analysis of the Data

The raw data supplied from IBM is displayed in Appendix A, section A1 and comprises of a table presenting the data for the five variables described above.

Initially, the data was explored using descriptive statistics. The aim of this part of the analysis was to quantify the data with numbers and to examine the data visually. However, in view of the (positive) skewness of Development effort, Maintenance effort, Total effort and System size it was decided to perform a logarithmic transformation of these variables (Coolican, 2009). When data are not normally distributed or are skewed it is recommended to transform the data (Barrow, 2001; Field, 2009). Data transformation changes the units of measurement of variables (but does not change the relative relationship between variables) and can render data suitable for analysis with parametric tests. The Log transformation (log base 10) was selected because it was considered suitable to reduce a positive skew and to stabilise the variance of the data (Field, 2009). Thus, variable Percentage of COTS components was not transformed because this variable was not skewed.
Descriptive statistics, including the Kolmogorov-Smirnov test of normality, Normal Q-Q Plots and histograms are presented for the raw data in Appendix A, Section A4. Normal Q-Q plots of the Log transformed variables are presented in Appendix A, Section A5.

Table 4.1 presents the summary statistics for both raw and transformed variables. The variables are grouped into Outcome and Predictor variables. These groupings were chosen because it was expected that Percentage of COTS components and System size would affect Development effort, Maintenance effort and Total effort.

<table>
<thead>
<tr>
<th>Item</th>
<th>Mean (Standard Deviation)</th>
<th>Median (Range)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Outcome variables</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Development Effort (DE)</td>
<td>3554 (5811)</td>
<td>1523 (34 – 44070)</td>
</tr>
<tr>
<td>Log(DE)</td>
<td>3.18 (.59)</td>
<td>3.18 (1.53 – 4.64)</td>
</tr>
<tr>
<td>Maintenance Effort (ME)</td>
<td>13626 (16868)</td>
<td>6620 (21 – 78121)</td>
</tr>
<tr>
<td>Log(ME)</td>
<td>3.71 (.77)</td>
<td>3.82 (1.32 – 4.89)</td>
</tr>
<tr>
<td>Total Effort (TE)</td>
<td>17180 (20185)</td>
<td>10150 (122 – 117533)</td>
</tr>
<tr>
<td>Log(TE)</td>
<td>3.91 (.62)</td>
<td>4.01 (2.09 – 5.07)</td>
</tr>
<tr>
<td><strong>Predictor variables</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Percentage of COTS components</td>
<td>52 (36)</td>
<td>50 (1 – 50)</td>
</tr>
<tr>
<td>System Size (SS)</td>
<td>5036 (9418)</td>
<td>1206 (20 – 53262)</td>
</tr>
<tr>
<td>Log(SS)</td>
<td>3.16 (.71)</td>
<td>3.08 (1.30 – 4.73)</td>
</tr>
</tbody>
</table>

Table 4.1: Summary statistics for n = 158 systems, where Log(xx) = measured on a logarithmic scale to base 10.

With reference to Table 4.1 it can be seen that for the 158 systems represented by the data Development effort, ranged from 34 to 44070 man hours. The mean value was 3554 man hours, the Standard Deviation 5811 and the median was 1523. Log(Development effort) ranged from 1.53 to 4.64. The Standard Deviation was .59. Both the mean and median values were 3.18, which indicate that the log transformation of the data corrected the skewness of this variable.
The raw data for *Maintenance effort* ranged from 21 to 78121 man hours and the Standard Deviation was 16868. The mean value was 13626 and the median was 6620. Log(Maintenance effort) ranged from 1.32 to 4.89, Standard Deviation was .77. The mean was 3.71 and the median was 3.82.

The raw data for *Total effort* ranged from 122 to 117533 man hours of effort. The Standard Deviation was 20185. The mean value was 17180 and the median 10150. Following the logarithmic transformation Log(Total effort) ranged from 2.09 to 5.07, Standard Deviation was .62, the mean was 3.91 and the median was 4.01.

Variable *Percentage of COTS components* ranged from 1 percent to 100 percent. The Standard Deviation was 36, the mean was 52 and the median 50. *Percentage of COTS components* did not follow a skewed distribution so this variable was not transformed.

*System size* ranged from 20 to 53262 Function Points. The Standard Deviation was 9418. The mean value was 5036 and the median 1206. Log(System size) ranged from 1.30 to 4.73. Standard Deviation was .71, mean 3.16 and median 3.08.

With reference to *System size* measured with Function Points, Software Measurement Services Ltd (2005) attempted to put this into context, stating that software size was not appreciated by non-technical people when expressed in Function Points and that expressing system size in terms of categories, such as, small, medium or large was more intuitive and convenient. Therefore, with reference to Software Measurement Services Ltd (2005) categories the *system size* data supplied by IBM ranged from *Extra small* (XS) through to *Extra-extra-extra large* (XXXL). With reference to IBM’s system size categories the data ranged from Very small through to Very large (Peter Thomas, IFPUG Certified Function Point Specialist, personal communication, April 01, 2008). A table documenting system size categories and their equivalent Function Point values can be found in Appendix A, Section A2. A table detailing IBM project size categorisation, measured in Function Points, is also provided in Appendix A, Section A2.

The next stage of the analysis involved investigating the relationships between the variables.
Coolican (2009) stated that before conducting correlation or regression analysis it was essential to plot a scatter diagram in order to visually examine the shape of the data. Therefore, the following section presents a set of scatter diagrams, the purpose of which were to determine if any relationships existed between pairs of the variables.

### 4.7 Scatter Diagrams Presenting the Data

Scatter diagrams are presented below to show the relationships between different pairs of variables. Variables Log(Development effort), Log(Maintenance effort), Log(Total effort) and Log(System size) are used. The raw data for Percentage of COTS components is used.

For information purposes scattered diagrams presenting plots of the raw data can be found in Appendix A, Section A6.

![Scatter diagram](image)

**Figure 4.1:** Scatter diagram plotting Percentage of COTS components with Log(Development effort).

Figure 4.1 was selected to determine if there was evidence to support the CBS Functional Density Rule of Thumb (Abts, 2002; 2004).
With reference to Figure 4.1 it can be seen that no obvious relationship (linear or non-linear) exists between variables *Percentage of COTS components* and Log(Development effort). Variable *Percentage of COTS components* was defined as the independent variable because Abts (2002; 2004) suggested that development effort should *reduce* as the percentage of system functionality supplied from COTS components *increased*. However, this effect was not noticeable with reference to Figure 4.1. The line of best fit, which was computed by PASW (‘add fit line at total’ was selected), only suggests a very weak negative association between the two variables. The R squared value shows that only 0.3% of the variation in Log(Development effort) can be attributed to *Percentage of COTS components*. The correlation coefficient $(r = -.052)$ confirms that the relationship between the two variables is not significant (See Section 4.7.1, point 1 for the result of the test of correlation using Pearson’s Product Moment Correlation).

![Figure 4.2: Scatter diagram plotting variables Percentage of COTS components and Log(Maintenance effort).](image)

Variables *Percentage of COTS components* and Log(Maintenance effort) were plotted to investigate whether any association existed between the percentages of functionality
supplied from COTS components and the effort to maintain systems over a five year period.

A visual inspection of Figure 4.2 does not suggest an obvious relationship between *Percentage of COTS components* and Log(Maintenance effort). The line of best fit only indicates a very weak negative association between the variables. The R squared value shows that only 0.2% of the variation in Log(Maintenance effort) can be attributed to *Percentage of COTS components*. The correlation coefficient ($r = -.044$) confirms that the relationship between the two variables is not significant (See Section 4.7.1, point 2 for the result of the test of correlation using Pearson’s Product Moment Correlation).

Thus, the scatter diagram did not suggest that an increase in *Percentage of COTS components* was associated with an obvious reduction in Log(Maintenance effort). However, maintenance effort figures were only available for a five year period.

Abts (2002; 2004) proposed that to *minimise* system maintenance costs systems should be built from as few COTS components as possible. However, with reference to Figure 4.2 it was not possible to comment on this proposal because details of the number of components contained within each system were not available.

Figures 4.3 and 4.4 are scatter diagrams presenting plots of Log(System size) with Log(Development effort) and Log(Maintenance effort) respectively. The purpose of Figures 4.3 and 4.4 were to investigate whether relationships existed between system size measured in Function Points and development and maintenance effort respectively. With reference to Figure 4.3 there was not a strong linear relationship between the two variables. The line of best fit displayed in Figure 4.3 suggests that the association between Log(System size) with Log(Development effort) is positive. However, the R squared value shows that only 4% of the variation in Log(Development effort) can be attributed to Log(System size). The correlation coefficient ($r = .201$) shows that there is a significant positive correlation between Log(Development effort) and Log(System size). However, the correlation is very weak (See Section 4.7.1, point 3 for the result of the test of correlation using Pearson’s Product Moment Correlation).
Figure 4.3: Scatter diagram plotting Log(System size) with Log(Development effort).

Figure 4.4 indicates a stronger relationship between Log(System size) and Log(Maintenance effort), compared with that displayed in Figure 4.3. This is reflected by the R squared value which shows that 17.9% of the variation in Log(Maintenance effort) can be attributed to Log(System size). The line of best fit confirms a positive association between the two variables. The correlation coefficient ($r = .423$) confirms that there is a significant positive correlation between Log(Maintenance effort) and Log(System size). However, the correlation is weak, but stronger than the correlation between Log(Development effort) and Log(System size) (See Section 4.7.1, point 4 for the result of the test of correlation using Pearson’s Product Moment Correlation).
Figure 4.4: Scatter diagram plotting Log(System size) with Log(Maintenance effort).

It could be valid to remove any outliers from the data if their removal does make a difference to the analysis (Grimm, 1993). However, with reference to Figures 4.3 and 4.4 and the number of outlying values there was difficulty in justifying which outlying values could be removed. Furthermore, the fact that there were numerous extreme values was interesting in itself as the expected trend was for development and maintenance effort to increase in accordance with an increase in the size of a system measured in Function Points; this association was not strong.

The Function Point value was deemed to be related to effort (Boehm, Abts & Chulani, 2000). Furthermore, FPA was proposed as an accurate predictor of effort and in software development environments (Giombetti, Hangal, Preissing & Trindade, 2006). However, with reference to the data used in this study the lack of strong associations between the data points presented in Figures 4.3 and 4.4 failed to support FPA as an accurate predictor of effort because effort did not increase in line with an increase in Function Points.
The purpose of Figures 4.5 and 4.6 are to present scatter plots of Log(Total effort) with Percentage of COTS components and Log(System size) respectively.

However, with reference to Figure 4.5 no obvious relationship between Log(Total effort) and Percentage of COTS components. This was to be expected because scatter diagrams presented in Figures 4.1 and 4.2 respectively failed to identify obvious relationships between Log(Development effort) and Percentage of COTS components and Log(Maintenance effort) and Percentage of COTS components; Log(Total effort) is the sum of Log(Development effort) and Log(Maintenance effort). The line of best fit suggests a slight negative relationship between Log(Total effort) and Percentage of COTS components. The R squared value shows that only 0.2% of the variation in Log(Total effort) can be attributed to Percentage of COTS components. The correlation coefficient \( r = -0.050 \) indicates that there is not a significant correlation between Log(Total effort) and Percentage of COTS components (See Section 4.7.1, point 5 for the result of the test of correlation using Pearson’s Product Moment Correlation).
Figure 4.6: Scatter diagram plotting Log(System size) with Log(Total effort).

As expected, there is a positive association between Log(System size) and Log(Total effort) displayed in Figure 4.6. The R squared value shows that 17.4% of the variation in Log(Total effort) can be attributed to Log(System size). There was a significant positive correlation between Log(Total effort) and Log(System size). The correlation coefficient \((r = .417)\) indicates that there is a significant positive correlation between Log(Total effort) and Log(System size) (See Section 4.7.1, point 6 for the result of the test of correlation using Pearson’s Product Moment Correlation).

For further information bar diagrams, displaying the raw values of Development effort, Maintenance effort and Total effort sorted by Percentage of COTS components and System size respectively can be found in Appendix A, Sections A3. These diagrams also reinforce the lack of obvious patterns between the variables.
4.7.1 Bivariate Correlation: Tests of Association using Pearson’s Product Moment

The purpose of this section is to present the results of the Pearson’s Product Moment Correlation. This was used to test the degree of association between pairs of variables.

A test of correlation was deemed relevant to examine the relationships between variables which were measured or collected simultaneously, but not manipulated, as in the case of an experiment (Coolican, 2009). This was the case with the system data provided from IBM because the variables were provided at the same time and were not manipulated during the research.

Additionally, correlation is the measurement of the extent to which pairs of related values on two variables tend to change together or ‘co-vary’ (Coolican, 2009). Therefore, a test of correlation was deemed appropriate to explore the relationships between pairs of the following variables: Development effort, Percentage of COTS components, System size, Maintenance effort and Total effort (Note: the logarithmic transformation of Development effort, System size, Maintenance effort and Total effort were used).

1) There was no significant correlation between Log(Development effort) and Percentage of COTS components.

\[(r = -.052, N = 158, p = .513, \text{two-tailed}).\]

2) There was no significant correlation between Log(Maintenance effort) and Percentage of COTS components.

\[(r = -.044, N = 158, p = .580, \text{two-tailed}).\]

3) There was a significant positive correlation between Log(Development effort) and Log(System size).

\[(r = .201, N = 158, p < .05, \text{two-tailed}).\]
4) There was a significant positive correlation between Log(Maintenance effort) and Log(System size).

\( r = .423, N = 158, p < .01, \) two-tailed).

5) There was no significant correlation between Log(Total effort) and Percentage of COTS components.

\( r = -.050, N = 158, p = .536, \) two-tailed).

6) There was a significant positive correlation between Log(Total effort) and Log(System size).

\( r = .417, N = 158, p < .01, \) two-tailed).

The following section presents the results of the multiple regression analysis.

**4.8 Multiple Regression Analysis**

Multiple regression models were used to test Hypotheses 1, 2 and 3. Multiple regression is a valid test when more than one predictor variable is believed to influence the dependant variable (Coolican, 2009), as was the case here. *Percentage of COTS components* and *System size* were considered to influence *Effort* (Development, Maintenance and Total effort).

The results of Hypotheses 1, 2 and 3 are presented below:

**4.8.1 Test of Hypothesis 1**

**Hypothesis 1**: *System size* and *Percentage of COTS components* are significant predictors of *Development effort*.

**Null Hypothesis 1**: *System size* and *Percentage of COTS components* are not significant predictors of *Development effort*. 
A multiple linear regression model was used to test Hypothesis 1. Log(Development effort) was defined as the dependent variable. *Percentage of COTS components* and Log(System size) were defined as the predictor variables.

Using Multiple Regression Analysis a significant multiple linear regression model emerged as p < .05 (p = .035) (see the ANOVA table in Table A7a in Appendix A, Section A7.1).

Only 4.2% of the variation of Log(Development effort) could be attributed to Log(System size) and *Percentage of COTS components* (see the model summary table produced by PASW in Table A7b in Appendix 7, Section A7.1).

Table 4.2, the Coefficients table which was produced by PASW, shows that *Percentage of COTS components* was not a significant predictor of Log(Development effort), but that Log(System size) was a significant predictor.

Note: the fitted regression equation, which is calculated from the Unstandardized coefficients, B column from Table 4.2, is presented in Appendix A, Section A7.1.

<table>
<thead>
<tr>
<th>Source</th>
<th>Unstandardised Coefficients</th>
<th>Standardised Coefficients</th>
<th>T</th>
<th>Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Constant)</td>
<td>2.692</td>
<td>.225</td>
<td>11.985</td>
<td>.000</td>
</tr>
<tr>
<td>Percentage of COTS components</td>
<td>-6.844</td>
<td>.001</td>
<td>-.042</td>
<td>.596</td>
</tr>
<tr>
<td>Log(System size)</td>
<td>.165</td>
<td>.065</td>
<td>.199</td>
<td>.013</td>
</tr>
</tbody>
</table>

*Table 4.2: Table of Coefficients for dependent variable Log(Development effort).*

The null hypothesis could not be rejected because *Percentage of COTS components* was not found to be a significant predictor of *Development effort*.

### 4.8.2 Test of Hypothesis 2

**Hypothesis 2:** *System size* and *Percentage of COTS components* are significant predictors of *Maintenance effort.*
**Null Hypothesis 2:** System size and Percentage of COTS components are not significant predictors of Maintenance effort.

To test Hypothesis 2 the following multiple linear regression model was used: Log(Maintenance effort) was defined as the dependent variable. Percentage of COTS components and Log(System size) were defined as predictor variables.

A significant multiple linear regression model emerged as $p < .05$ ($p = .000$) (see the ANOVA table produced by PASW in Table A7d in Appendix A, Section A7.2).

Only 18% of the variation of Log(Maintenance effort) could be attributed to Log(System size) and Percentage of COTS components. Thus, 82% of the variation in Log(Maintenance effort) is explained by other factors (see the model summary table, produced by PASW, in Table A7e in Appendix 7, Section A7.2).

The Coefficients table produced by PASW and presented in Table 4.3 shows that Percentage of COTS components was not a significant predictor of Log(Maintenance effort). However, Log(System size) was found to be a significant predictor of Log(Maintenance effort).

The fitted regression equation which emerged from this analysis is presented in Appendix A, Section A7.2.

<table>
<thead>
<tr>
<th>Source</th>
<th>Unstandardised Coefficients</th>
<th>Standardised Coefficients</th>
<th>t</th>
<th>Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B</td>
<td>Std. Error</td>
<td>Beta</td>
<td></td>
</tr>
<tr>
<td>(Constant)</td>
<td>2.294</td>
<td>.271</td>
<td>8.454</td>
<td>.000</td>
</tr>
<tr>
<td>Percentage of COTS components</td>
<td>-4.679</td>
<td>.002</td>
<td>-.301</td>
<td>.764</td>
</tr>
<tr>
<td>Log(System size)</td>
<td>.456</td>
<td>.079</td>
<td>5.793</td>
<td>.000</td>
</tr>
</tbody>
</table>

Table 4.3: Table of Coefficients for dependent variable Log(Maintenance effort).

The null hypothesis could not be rejected because Percentage of COTS components was not found to be a significant predictor of Maintenance effort.
4.8.3 Test of Hypothesis 3

**Hypothesis 3:** System size and *Percentage of COTS components* are significant predictors of Total effort.

**Null Hypothesis 3:** System size and *Percentage of COTS components* are not significant predictors of Total effort.

To test Hypothesis 3 the following multiple linear regression model was used: Log(Total effort) was defined as the dependent variable. *Percentage of COTS components* and Log(System size) were defined as predictor variables.

A significant multiple linear regression model emerged as p < .05 (p = .000) (see the ANOVA table produced by PASW in Table A7g in Appendix A, Section A7.3).

Only 17.5% of the variation of Log(Total effort) could be attributed to Log(System size) and *Percentage of COTS components*. Thus, 82.5% of the variation in Log(Total effort) is explained by factors not included in this model (see the model summary which was produced by PASW in Table A7e in Appendix 7, Section A7.3).

Table 4.4, the Coefficients table produced by PASW, shows that *Percentage of COTS components* was not a significant predictor of Log(Total effort), but that Log(System size) was a significant predictor.

The fitted regression equation which emerged from this analysis is presented in Appendix A, Section A7.3.

<table>
<thead>
<tr>
<th>Source</th>
<th>Unstandardised Coefficients</th>
<th>Standardised Coefficients</th>
<th>t</th>
<th>Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B</td>
<td>Std. Error</td>
<td>Beta</td>
<td></td>
</tr>
<tr>
<td>(Constant)</td>
<td>2.787</td>
<td>.220</td>
<td>12.644</td>
<td>.000</td>
</tr>
<tr>
<td>Percentage of COTS components</td>
<td>-4.747</td>
<td>.001</td>
<td>-.027</td>
<td>-.376</td>
</tr>
<tr>
<td>Log(System size)</td>
<td>.363</td>
<td>.064</td>
<td>.415</td>
<td>5.684</td>
</tr>
</tbody>
</table>

Table 4.4: Table of Coefficients for dependent variable Log(Total effort).
The null hypothesis could not be rejected because *Percentage of COTS components* was not found to be a significant predictor of *Total effort*.

### 4.9 Discussions of the Results

The aim of this section is to discuss the results of the statistical analysis of the data provided from IBM.

One claimed benefit of CBD was reduced development costs (Voas, 1998; Leung & Leung, 2002). However, Figure 4.1, a scatter diagram plotting *Percentage of COTS components* with Log(Development effort), did not indicate an obvious association between the two variables. Thus, with reference to the claimed premise that CBD should result in reduced development costs it could be suggested that development effort for systems built from greater percentages of COTS components should be lower than systems built from lower percentages of COTS components. However, this relationship was not seen in Figure 4.1.

Furthermore, the R Squared value, displayed in Figure 4.1, indicated that only 0.3% of the variation of Log(Development effort) could be explained by *Percentage of COTS components*.

The multiple linear regression model used to test Hypothesis 1 indicated that *Percentage of COTS components* was not a significant predictor of Log(Development effort) and that only 4.2% of the variation of Log(Development effort) could be attributed to Log(System size) and *Percentage of COTS components*. Thus, 95.8% of the variation in Log(Development effort) was explained by factors not included in this model.

The results of the test of Hypothesis 1 identified a limitation of using a multiple linear regression analysis; there may be additional *unmeasured* predictor variables which may have resulted in a greater proportion of the variance of the predictor variable.

Therefore, by their very nature, because the ‘missing variables’ were not collected it was not possible to assess their contribution to the variation of the predictor variable within this analysis (Brace et al., 2006; Coolican, 2009). Therefore, an area for further
research is to identify additional predictor variables which would account for the variation of system development effort.

However, the lack of association between Log(Development effort) and Percentage of COTS components was itself an interesting finding because this did not support the CBS Functional Density Rule of Thumb (Abts, 2002; 2004). With reference to the CBS Functional Density Rule of Thumb (Abts, 2002; 2004) it was expected that Log(Development effort) would decrease for systems where system functionality was supplied from higher percentages of COTS components. However, this was not reflected in this analysis.

However, Percentage of COTS components did not include details, such as, the type and number of components contained within each system, the amount of glue code or wrapper code (if relevant) or the component configuration and integration effort performed. Thus, it was not possible to ascertain the types of activities represented by Development effort.

Another interesting finding concerned the weak association between Log(Development effort) and Log(System size) suggested in Figure 4.3. The R Squared value, displayed in Figure 4.3, indicated that only 4% of the variation in Log(Development effort) could be attributed to Log(System size). The test of Hypothesis 1 showed that although Log(System Size) was a significant predictor of Log(Development effort) its effect was poor. With reference to the claims that FPA is a reliable predictor of system development effort (Ahmed et al., 2008; Zheng et al., 2009), it was expected that the association between Log(Development effort) and Log(System size) would have been stronger. Thus, the result from this analysis suggests that further research should be conducted into the reliability of FPA as an accurate predictor of system development effort, especially for systems built from components.

Abts (2004) stated that Function Point measurements were originally calibrated against building systems from scratch and, as such, further work was required to assess the suitability of Function Points as a predictor of effort to deliver functionality supplied from components. This was supported by Wijayasiriwardhane and Lai (2008), who stated that although FPA is applicable for predicting the effort required to develop glue-
code for component integration no extension of FPA is proposed for COTS-based systems at the system level due to the black-box nature of components.

With reference to Wijayasiriwardhane and Lai (2008) it can be seen that the reliability of a system sizing metric, such as Function Points, can be questioned when used to both quantify the size of COTS-based systems and predict the effort required to develop systems as a whole. The reason for this is that the development effort to produce each component (and the corresponding functionality, as measured in Function Points) has already been incurred by the component vendor/developer in producing the component in the first place (this is the basis of the cost saving claim of CBD. The time and effort incurred by a vendor to produce a component results in a system developer saving time and cost by not having to develop the functionality provided by the component). However, system developers using COTS components to build systems incur effort performing different tasks, compared with developers producing custom-built systems. Although system developers may be required to produce glue code and wrapper code other tasks requiring effort which are specific to CBD include: identifying candidate components, assessing and testing component suitability, tailoring and configuring components, and integrating components to produce a completed system. Thus, the effort required to perform tasks specific to COTS-based systems may not be represented by the relative size of each component and other system parts, measured in Function Points.

However, Wijayasiriwardhane and Lai (2008) suggested that development effort and the size of COTS-based systems were related to the number of components integrated into each system. They suggested that systems containing greater numbers of components tended to be larger in size than systems comprising of fewer components. Thus, systems comprising of more components require more effort to acquire, tailor and integrate the additional number of components.

However, with reference to variables System size and Percentage of COTS components details of, for example, the types and number of components contained within each system, the size of components, the amount of glue code or wrapper code produced were not known. Furthermore, it was not known what tasks were represented by the Development effort figures or whether this was a consistent measurement across all of the systems included in the data set. It was also not known who performed the Function
Point Analysis for the systems, who recorded the data, or whether the same person recorded the data for all of the systems. Further research into this area should aim to establish comprehensive details about the nature of systems being studied.

The use of Function Points as an accurate predictor of *Maintenance effort* was not supported. For example, although the scatter diagram displayed in Figure 4.4 indicated that the degree of association between \( \log(\text{Maintenance effort}) \) and \( \log(\text{System size}) \) was stronger than the association between \( \log(\text{Development effort}) \) and \( \log(\text{System size}) \) the association was still weak. The R Squared value displayed in Figure 4.4 indicated that only 17.9% of the variation of \( \log(\text{Maintenance effort}) \) could be explained by \( \log(\text{System size}) \). This effect was supported from the results of the test of Hypothesis 2. Although, \( \log(\text{System size}) \) was found to be a significant predictor of \( \log(\text{Maintenance effort}) \) the combination of *Percentage of COTS* and \( \log(\text{System size}) \) only accounted for 18% of the variation of \( \log(\text{Maintenance effort}) \). Thus, 82% of the variation in \( \log(\text{Maintenance effort}) \) was explained by other factors.

Several system development and maintenance effort estimation models exist which reference Function Points as a software size metric. COCOMO and the Annual Change Traffic Model are examples of effort estimation models (Ahn, Suh, Kim & Kim, 2003). However, these models do not consider systems built from components.

Therefore, further research is required to investigate the relationships between system and component Function Point size measurements and the effort required to perform system development and maintenance tasks for COTS-based systems.

The results of the tests of Hypotheses 2 and 3 showed that variable *Percentage of COTS components* was not found to be a significant predictor of \( \log(\text{Maintenance effort}) \) and \( \log(\text{Total effort}) \) respectively.

Abts (2002, p. 5) claimed, in the *CBS Functional Density Rule of Thumb*, that to “reduce maintenance costs COTS-based systems should be constructed from as few COTS components as possible”. However, it was not possible to comment on this from the analysis because the data did not include details of the number of components used to construct each system. An area of further research would be to collect metrics on the number of components included within each system.
Furthermore, Reifer, et al., (2003) suggested that the costs to maintain COTS-based systems may equal or exceed the maintenance costs of custom software. However, the results of the tests of Hypotheses 2 and 3 did not support or refute this because there were no indications that the Maintenance effort or Total effort values of systems built from greater proportions of COTS components were higher, compared with systems built from lower percentages of COTS components. Further to this, the scatter diagrams displayed in Figures 4.2 and 4.5 did not reveal obvious associations between Percentage of COTS components and Log(Maintenance effort) and Log(Total effort) respectively.

A further limitation of this analysis was that the data did not include details of the maintenance tasks performed over the five year period, which were represented by the Maintenance effort and Total effort figures. However, it can be seen that increasing maintenance costs may not always be an unwelcome consequence; it is dependant upon who is responsible for bearing the cost of maintenance activity. For example, when a customer requests a component upgrade in order to benefit from additional functionality this may be viewed as beneficial for the system management company as it may constitute a revenue generation opportunity. Conversely, component maintenance activity, such as security patching, which may be forced upon system administrators by vendors may be viewed as an additional cost for system administrators if the contractual obligation classifies this activity as being under their responsibility.

Another issue with the Maintenance effort and Total effort variables were that these figures only represented maintenance activity over a five year period. Thus, it was not known how long the systems were planned to be in service. Furthermore, it could be suggested that in some cases, higher maintenance costs per year may be acceptable for systems which are planned to be in service for many years as, from a cost perspective, the continued maintenance costs may be less than the costs to develop a new system. However, for systems with shorter planned lifecycles higher yearly maintenance costs may not be acceptable.

4.10 Conclusions

This section provided the analysis of data provided by IBM, which comprised of the following variables:
• **Development effort** (System development effort measured in man hours);
• **Percentage of COTS components** (Percentage of system functionality provided from COTS components);
• **System size** (System size measured in Function Points);
• **Maintenance effort** (System maintenance effort over 5 year period measured in man hours);
• **Total effort** (Total system life effort (Development effort plus Maintenance effort) measured in man hours).

The preliminary analysis of the variables suggested that a logarithmic transformation of **Development, Maintenance, Total effort** and **System size** should be performed in view of the (positive) skewness of the data.

Scatter diagrams plotting pairs of the variables showed that no apparent linear relationships existed when **Percentage of COTS components** was plotted with Log(Development effort), Log(Maintenance effort) and Log(Total effort).

The scatter plots of Log(System size) with Log(Development effort), Log(Maintenance effort) and Log(Total effort) did not reveal strong relationships between the variables.

To test Hypothesis 1 a multiple linear regression analysis indicated that Log(System size) was a significant predictor of Log(Development effort). However, **Percentage of COTS components** was not. Furthermore, this multiple linear regression model only explained 4.2% of the variance of Log(Development effort).

Multiple linear regression analysis indicated that Log(System size) was a significant predictor of Log(Maintenance effort). However, **Percentage of COTS components** was not. Furthermore, this multiple linear regression model only explained 18% of the variance of Log(Maintenance effort). This was used to test Hypothesis 2.

A multiple linear regression model was used to test Hypothesis 3. This showed that Log(System size) was a significant predictor of Log(Total effort). However, **Percentage...
of COTS components was not. This multiple linear regression model only explained 17.5% of the variance of Log(Maintenance effort).

This analysis has suggested that, in addition to variables Percentage of COTS components and System size, factors other than those available for statistical analysis, contributed to the variance of Development effort, Maintenance effort and Total effort.

Thus, the purpose of the following chapter is to present the results of Grounded Theory analysis, which aims to identify other factors which contribute to the cost of building systems from COTS components.
Chapter 5 Grounded Theory Analysis

5.1 Introduction

The aim of the Grounded Theory analysis was to investigate the cost factors of CBD using data collected from within IBM. Interview data was collected from software development practitioners who had worked within the CBD domain. It should be noted that it was hoped that documentary data, such as design documents, meeting minutes and emails relating to COTS-based projects would also be analysed. However, no documentary data was provided.

The Grounded Theory method was used to analyse the data. The justification for the choice of the interview data collection method is described in Section 3.5.1.

5.2 Choice of Research Subjects

The aim of the Grounded Theory analysis was to target system, project and software practitioners involved with COTS-based system development from within IBM, a large multinational corporation. From a global perspective IBM was involved in many facets of Information Technology (IT). The company’s interests ranged from IT equipment manufacturing, software design and development through to business innovation. Furthermore, commercial confidentiality prevented the disclosure of specific commercial details about practitioner involvement. However, this also afforded a freedom in reporting this research as the anonymity allowed adverse as well as favourable points to be made thereby promoting an unbiased view of research results (Allan, 2003).

Practitioner selection was based upon the following criteria:

- They had been involved in COTS-based development. The criteria for involvement covered a wide range of roles, including project management, IT Architect and software development.

- They were willing to participate in the study.
The recommended method to reduce bias in a study was for the research subjects to be selected by random sampling (Kitchenham, 2002). This was a technique where subjects (a sample) were randomly selected from a larger group (a population). Each individual was chosen entirely by chance and each member of the population had an equal chance of being included in the sample (Robson, 2003).

However, the software practitioners identified to be interviewed were not selected by random sampling for the Grounded Theory analysis. The main reason for this concerned the limited availability of suitable interviewees. A population of software practitioners dealing solely with COTS-based development and maintenance did not exist within the company. The company’s software development and maintenance structure resulted in COTS-based software practitioners working on a variety of projects within different parts of the company. The number of identified practitioners was limited to a small group. It was felt that to further select a random sample from this group would not add value to this study. Furthermore, the Grounded Theory method, which was used to analyse the interview data, treated selection bias as just another variable which will emerge as a result of conceptualisation (Glaser, 2001).

Interviewee selection was initiated by sending an email request to senior IT architects within the company. They were asked to recommend appropriate contacts involved with CBD. These contacts were approached and then asked to suggest additional contacts who were contacted in the same manner.

A variation on the above method, which was used to identify further interviewees, was where all interviewees were asked who else would be worthwhile to interview (Walsham, 1994). This method was deemed valid because it was felt that COTS-based software practitioners would be likely to know other practitioners working in the same field.

The first interview was with a Project Manager. A Project Manager was considered to be the central figure within each project and thus, would be best placed to recommend further interviewees.

In total, eleven practitioners were interviewed. A summary of the interviewees can be found in Section 5.4.
5.3 Data Collection

An important reason for selecting the interview data collection method was that it was feasible to perform. As mentioned earlier, the selected interviewees were willing to be interviewed. Furthermore, many of the participants informally expressed a preference to being interviewed, rather than to being asked to fill in a questionnaire or complete a survey. It was felt that for people who were very busy being interviewed in an informal setting was less demanding on their time.

Yin (2003) recommended that in order to exploit the flexibility of the case study method interviews should take the form of a guided conversation rather than a structured query. An interview which was too structured would have more similarities to a ‘verbal’ questionnaire or survey and not offer the flexibility which was originally desired.

Glaser (1978), however, was not clear on the level of structure which should be applied to data collection. Glaser (1978, p. 44) suggested that the researcher should approach data collection “with complete openness”. However, Glaser (1978, p. 45) implied that some form of structure was acceptable. He stated that “researchers find it more comfortable to enter the field with some combination of a clear question or problem area in mind….This is less than being completely open”.

The semi-structured interview technique was used as the data collection method because it enabled the subjects to offer interpretations of the research issues in their own words and to allow the interviewees to express their view of the world (Robson, 2003). It was also felt that a certain degree of structure was required to keep each interview within the scope of the research area. The semi-structured interview was considered to satisfy this condition (Robson, 2003).

Each interview commenced with the interviewer asking a couple of key open ended questions. The purpose of this approach was to encourage the interviewee to talk freely around the subject area, rather than constraining the interviewee by applying too much structure to the interview (Miles & Huberman, 1994).

The interviews were performed either face to face or over the telephone.
Furthermore, each interview was limited to between 30 minutes and one hour in length. This decision was based upon several factors:

- Due to the nature of the business all of the participants were very busy. Interview sessions of this length appeared long enough for the researcher to gain sufficient detailed data, whilst being short enough to gain interviewee participation. In some cases interview sessions took place in a quiet corner of the canteen and thus doubled as a break for the interviewee. In other cases, due to the location of some participants, interviews were conducted over the telephone.

- Interview sessions of up to an hour fitted in well with the structure of the business day. Other internal meetings tended to be scheduled in one hour segments. For face to face meetings, the meeting room booking procedure also followed this time format.

- Attention span limitations were considered. It was found that the attention span of both interviewer and interviewee waned after an hour.

The interview data collection process followed Glaser’s (1978) guidelines and was performed as follows:

1) The process commenced by conducting and analysing (coding) the first interview; the results of this analysis served to define the focus of the next interview and so on.

2) The details arising from the interviews were recorded as handwritten field notes during the course of each interview and typed up following the interview. This process was applied to both face to face and telephone interviews.

3) As an aid to the organisation of interview data Miles and Huberman (1994) recommended producing a Contact Summary Sheet following each interview. This format formed the basis of the write-up of the field notes.

On recording the output of interviews Glaser (1998, p. 107) stressed that interviews “should not be taped”. His justification was that the tape recording of interviews would result in too much data and the process of transcribing the tapes would take too long,
thus, detracting from the spontaneity of the analysis process and stifle the interviewer’s creativity in interpreting the data.

However, this differed from other proponents of the interview data collection method who recommended the use of some form of voice recording device to make a record of the proceedings (Miles & Huberman, 1994; Charmaz, 2006). Their justification was that this allowed the researcher to revisit the data many times.

5.4 Summary of Practitioners Interviewed

This section provides a summary of the eleven interviewees. The key text identifier assigned to each interview is also provided. The purpose of key text identifiers are explained in Section 5.5.

Interview A

The participant of Interview A was classified within IBM as a Project Manager. Project Managers are responsible for the implementation and management of projects ranging from system implementation through to system maintenance tasks such as component upgrades and patching. Besides the knowledge of methods to initialize, plan, execute, control and monitor a project a project manager needs social skills and leadership competencies, starting with communication basics throughout team building, negotiation skills, decision making, problem solving and conflict management.

The interviewee was based in the United States had thirteen years experience with the management of both custom and COTS-based system projects.

Key text data from this interview was assigned code PMiAnn, where ‘nn’ is the key text identifier.

Interview B

Interviewee B was an IT Architect. Within IBM IT architects define solutions to client business problems through the reasoned application of information technology. IT architects are required to possess a broad skill base and experience in multiple systems, platforms, operations, infrastructure and application aspects and design techniques. Interviewee B has worked for IBM in the United Kingdom for eighteen years and has been involved with designing large custom and COTS-based solutions.
Interview C
Interviewee C was an IT Architect and had worked for IBM in the United Kingdom for ten years. Over the last five years Interviewee C has been involved with designing solutions involving the configuration and integration of large COTS packages.

Key text data from this interview was assigned code ARiBnn, where ‘nn’ is the key text identifier.

Interview D
The job description of Interviewee D was a Test Architect. Within IBM a Test Architect is part of the IT Architect profession. However, the focus of a Test Architect is to design test cases whereby solutions can be tested. Interviewee D has performed this role within IBM in the United Kingdom for seven years and has been involved in the testing of both custom and COTS-based applications and systems.

Key text data from this interview was assigned code ARiCnn, where ‘nn’ is the key text identifier.

Interview E
Interviewee E was a software developer, based in the United Kingdom. He has been working on a large IBM strategic outsourcing project for the last seven years. The basis of the project has been to develop and manage a large financial system of behalf of a customer. The system comprised of the integration of a custom-built application with numerous COTS components. His role involved planning and implementing the integration and maintenance of the COTS components relating to this system.

Key text data from this interview was assigned code ARiDnn, where ‘nn’ is the key text identifier.
Interview F
Interviewee F was an Application Architect. This role is part of the IBM IT Architect profession. The focus of an Application Architect was to design applications. The assumption was that the underlying system infrastructure has already been designed. Interviewee F has worked for seven years and had focused upon designing COTS-based applications.

Key text data from this interview was assigned code ARiFnn, where ‘nn’ is the key text identifier.

Interview G
Interviewee G was an IBM Project Manager, based in Philadelphia, United States. He has worked for IBM for seventeen years and has managed numerous custom and COTS-based projects. His last project involvement was the management of the development and implementation of a large COTS-based financial application.

Key text data from this interview was assigned code PMiGnn, where ‘nn’ is the key text identifier.

Interview H
Interviewee H was a Project Manager, based in Tuson, Arizona in the United States. He has worked for IBM for seventeen years and has managed projects for global customers involving the implementation of custom and COTS-based systems.

Key text data from this interview was assigned code PMiHnn, where ‘nn’ is the key text identifier.

NOTE: ‘Interview I’ was not used as it was felt that the letter ‘I’ could get confused with a ‘1’ in the coding scheme.

Interview J
Interviewee J was an IBM Project Manager, based in Raleigh, North Carolina in the United States. He has worked for IBM for ten years and had managed small and large projects involving both custom and COTS-based technologies.
Key text data from this interview was assigned code PMiJnn, where ‘nn’ is the key text identifier.

**Interview K**
Interviewee K was an IT Architect who had worked for IBM in the United Kingdom for twelve years. He was involved with the design of custom and COTS-based applications and systems, mainly for the financial sector.

Key text data from this interview was assigned code ARiKnn, where ‘nn’ is the key text identifier.

**Interview L**
Interviewee L was an Integration Architect. He was involved with designing solutions to enable the integration of custom-built and COTS-based system components. He had worked for IBM in the United Kingdom for ten years.

Key text data from this interview was assigned code ARiLnn, where ‘nn’ is the key text identifier.

### 5.5 Overview of the Interview Data Analysis Process

This section explains the Grounded Theory interview data analysis process. This process began following the first interview. The Grounded Theory method (Glaser, 1978) prescribed that the collection and analysis of interview data should be an iterative process. The results from the analysis of the first interview determined the focus of the second and subsequent interviews.

The interview data analysis process is summarised in Figure 5.1.

The process involved examining the field notes line by line to identify *key text*. Key text was defined as items of text which were considered to be relevant to the investigation. The key text was added to a table and assigned a *key text identifier* (ID), such as PMiA1. The following naming convention was used for the key text code - the first two characters identified the professional area of the data source.
For example, for the first interview, ‘PM’ referred to *Project Manager*. The next character identified the data source type, ‘i’ for *interview*. A character to identify the data source followed. For example, an ‘A’ referenced *Interview A*. The last number identified the *key text* selection and increased sequentially. The coding scheme enabled each piece of *key text* to be traced back to the original data source.

The content of each piece of key text was analysed by exploring the words and phrases used by the interviewees to encompass the underlying principles in *Open Codes* (Glaser, 1978). The open coding process is defined in Section 3.6.3.1.

In conjunction, open codes were compared to identify underlying patterns to suggest emerging concepts. The concepts were also compared with each other to identify common characteristics which led to the identification of *categories*. This process was known as *Constant Comparison* (Glaser, 1978) and is described in Section 3.6.3.2.

Glaser (1978) stated that it is from the discussion of the relationships between the categories that the formation of a Grounded Theory should emerge.

An example of the analysis of the first interview is presented in section 5.6. This will be followed by the summary of the analysis of subsequent interviews.
5.6 Analysis of Interview A

The participant of Interview A was classified within IBM as a Project Manager. Project Managers are responsible for the implementation and management of projects ranging from system implementation through to system maintenance tasks such as component upgrades and patching. The interviewee had thirteen years experience with the management of both custom and COTS-based system projects.

5.6.1. Open Coding

A recommendation to aid coding at the conceptual level was to attach meaningful gerunds (a gerund is a verbal noun usually ending in –ing) to the identified issue in the data (Allan, 2003). An alternative suggestion was to use the term ‘naming’ for this stage of the Grounded Theory analysis; the aim being to attribute a name to a key point in the data which represented an interpretation of what was happening (Locke, 2005).

The selection of an appropriate data portion size for further examination proved challenging. Strauss and Corbin’s (1998) Grounded Theory approach prescribed that the researcher first performs a microanalysis of the data involving the close examination of each line, word, sentence or paragraph. However, the close examination of very small chunks of the data, such as each word, was found to be too time consuming.

Glaser (1978) and Charmaz (2006) suggested that a manageable data chunk for analysis was a sentence. A sentence was defined as a grammatical unit normally containing a subject, a verb and an object (Seely, 2004). However, in interview field notes it could be difficult to tell where one sentence ended and another began.

The open coding process for Interview A involved assigning ‘Open Codes’ to ‘key points’ of text taken from the interview field notes. The aim was to assign ‘conceptual’ codes which represented the underlying process of the key text. For example, in key text PMiA1 the interviewee stated that COTS components were reused in a particular application. This was assigned open code ‘System developers using COTS components’ because the interviewee implied that system developers had made the choice to use COTS components in the application. Thus, the underlying process of this code referenced the strategy of choosing to use COTS components instead of employing other development methods. The next point in the interview was key text PMiA2, which
was: ‘considered this to have saved 600 man hours in initial development time’. The open code assigned to key text PMiA2 was ‘Saving development time’ because, from the interview data, the use of COTS components were deemed to have saved development time. Therefore, by reusing COTS components system developers were saving development time.

Several issues with the method of coding were identified. Preconceived codes should not be forced onto the data (Glaser, 1978; Charmaz, 2006). However, it was challenging to dissociate pre-conceived themes from the data, especially if challenging taken-for-granted ideas. Therefore, it was important to ensure that all codes ‘earned’ their way into the analysis of the data.

Clustering diagrams were constructed to translate from words to diagrams and assist the investigation of relationships between some of the categories, causes and effects. By drawing a circle around a central code or idea the aim was to explore the relationship, in diagrammatical form, between open codes, concepts and categories. Other codes or ideas were linked to the central theme by lines. Where relevant, labels were added to the lines to add meaning (Charmaz, 2006) Clustering diagrams were found to be beneficial because they offered a method of developing a visual representation of the relationship between open codes and concepts.

Table 5.1 lists the key text selections, key text IDs and open codes resulting from the analysis of Interview A.

<table>
<thead>
<tr>
<th>CodeNo.</th>
<th>Key Text</th>
<th>Open Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMiA1</td>
<td>[COTS components] used in this application</td>
<td>System developers using COTS components</td>
</tr>
<tr>
<td>PMiA2</td>
<td>considered this to have saved 600 man hours in initial development time</td>
<td>Saving development time</td>
</tr>
<tr>
<td>PMiA3</td>
<td>the maintenance of multiple interfaces takes much more effort and planning</td>
<td>Maintaining multiple interfaces</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Increasing system maintenance effort</td>
</tr>
<tr>
<td>PMiA4</td>
<td>Using 50% of the functionality of a large component, which solves 90% of all business problems</td>
<td>Preferring large components</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Solving business problems</td>
</tr>
<tr>
<td>PMiA5</td>
<td>is far better than utilising 100% of the functionality of numerous smaller components</td>
<td>Architects recommending using fewer components</td>
</tr>
<tr>
<td>CodeNo.</td>
<td>Key Text</td>
<td>Open Codes</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PMiA6</td>
<td>integrating more COTS components which have the same architectural basis</td>
<td>Selecting architecturally compatible components</td>
</tr>
<tr>
<td>PMiA7</td>
<td>is better than attempting to integrate fewer components which are architecturally different.</td>
<td>Avoiding architectural incompatibility</td>
</tr>
<tr>
<td>PMiA8</td>
<td>Architecturally disparate components require more integration effort.</td>
<td>Selecting architecturally incompatible components Integration effort</td>
</tr>
<tr>
<td>PMiA9</td>
<td>if they are supplied by different vendors.</td>
<td>COTS supplier issues</td>
</tr>
<tr>
<td>PMiA10</td>
<td>there may also be different support agreements and upgrade roadmap policies</td>
<td>Multiple vendors Conflicting maintenance schedules</td>
</tr>
<tr>
<td>PMiA11</td>
<td>The assessment of COTS components has been a major cost factor</td>
<td>Assessing component suitability Increasing system development costs</td>
</tr>
<tr>
<td>PMiA12</td>
<td>Although the functionality of components are published it can be very time consuming to determine if they are actually suitable.</td>
<td>Establishing component suitability Requiring effort</td>
</tr>
<tr>
<td>PMiA13</td>
<td>In some instances the terminology used to describe their functionality can be confusing</td>
<td>Difficulty understanding terminology Interpreting language</td>
</tr>
<tr>
<td>PMiA14</td>
<td>The maintenance hours have been less for this system than for other system types because of the way it was designed</td>
<td>Designing for change</td>
</tr>
<tr>
<td>PMiA15</td>
<td>the actual application did not require to be changed much</td>
<td>Designers minimising system change</td>
</tr>
<tr>
<td>PMiA16</td>
<td>There is also a difference in the understanding of what constitutes system maintenance</td>
<td>Problems classifying system maintenance</td>
</tr>
<tr>
<td>PMiA17</td>
<td>Some developers classify/incorporate system enhancement activity as system maintenance</td>
<td>Lacking common understanding of maintenance</td>
</tr>
<tr>
<td>PMiA18</td>
<td>dependencies which were not immediately apparent</td>
<td>Appreciating system dependencies</td>
</tr>
<tr>
<td>PMiA19</td>
<td>upgrading the underlying OS can cause the COTS products not to work correctly</td>
<td>System dependencies</td>
</tr>
</tbody>
</table>

Table 5.1: Key text, key text IDs and open codes from Interview A.

5.6.2 Emergence of Concepts from Interview A

The nomenclatures used in the text are as follows:

- Open codes are identified in italics and include the ‘Key text identifier;
- Concepts are identified in italics. The first letter of each word is in upper case;
- Categories are in upper case and highlighted in bold;
- The core category is identified in upper case, highlighted bold and underlined.
The open codes listed in Table 5.1, were analysed, using the constant comparative method (Glaser, 1978).

Constant comparison was performed using *Memos*. The use of memos was considered to be a crucial part of the analysis using the Grounded Theory method and permeated all aspects of the analysis (Glaser, 1978; 1998; Charmaz, 2006). Memos allowed for ideas to be explored in free writing and for an analyst to articulate interpretive thinking.

The first part of the constant comparative process was to group the open codes together with reference to a common theme, which was an abstract representation of a higher order of commonality. This common theme was referred to as a *concept*. Glaser (2001, p. 9) stated that conceptualisation was “the core process of Grounded Theory”. He defined concepts as being “abstract of time, place and people and that [those] concepts have enduring grab, which appeal can go on forever as an applied way of seeing events” (Glaser, 2001, p. 10). However, during the analysis it proved difficult to confirm if Glaser’s (2001) view on conceptualisation was being applied correctly.

Further concepts emerged from other groupings of the codes. Full details of the analysis of Interview A can be found in Appendix B1.

For example, Code “*System developers using COTS components*” emerged from key point *PMiA1*. This code was compared with the next code in Interview A, *Saving development time PMiA2*, to identify a shared theme. The shared theme was concept *Cost Reducing Strategy*, which was identified for the following reasons:

Building systems from COTS components was considered to save development time because software developers avoided the time spent producing the functionality supplied by the COTS software components; this was justified because the development effort had already been performed by COTS component developers. Therefore, the thought decisions underpinning code *System developers using COTS components PMiA1* could be seen to be related to concept *Cost Reducing Strategy* because system developers assumed that using COTS components would result in *Saving development time PMiA2*. Concept *Cost Reducing Strategy* was considered a ‘strategy’ because the choice to build systems from COTS components, as opposed to selecting other
development methods such as the custom-built approach, was consciously made. Glaser (1978) defined a strategy as a tactic or a means of dealing with events. Thus, the choice of System developers using COTS components PMiA1 could be seen as a conscious tactic employed by developers, with the aim of Saving development time PMiA2. Saving development time PMiA2 was assumed to contribute to reducing costs as a result of a saving in the cost of employing human resources.

Furthermore, the Cost reducing strategy of building systems from COTS components was be seen to support a ‘causal-consequence’ model (Glaser, 1978). With the comparison of codes System developers using COTS components PMiA1 and Saving development time PMiA2 a causal factor was the economic pressures experienced by IT companies related to continually striving to reduce system development costs. Thus, the consequence was System developers using COTS components PMiA1; the choice of a system development method promising development cost savings.

The comparisons of the following combinations of open codes from Interview A were also deemed to relate to concept Cost Reducing Strategy:

- Saving development time PMiA2 and Preferring large components PMiA4;
- Saving development time PMiA2 and Selecting architecturally compatible components PMiA6;
- Saving development time PMiA2 and Avoiding architectural incompatibility PMiA7;
- Saving development time PMiA2 and Multiple vendors PMiA10;
- Increasing system maintenance effort PMiA3, Selecting architecturally compatible components PMiA6, Avoiding architectural incompatibility PMiA7, Preferring large components PMiA4, Solving business problems PMiA4 and Architects recommending using fewer components PMiA5.

The concept of Balancing Cost Challenges also emerged from the comparison of codes from Interview A. From the data, Balancing Cost Challenges conceptualised the cost trade off actions performed by COTS-system practitioners when developing and maintaining COTS-based systems. From the literature, a proposed benefit of adopting the COTS-based approach was ‘reduced costs’ (Ballurio et al., 2002; Yang et al., 2005;
Li et al., 2006; Clark & Clark, 2007; Cook, 2007; Li et al., 2008). Thus, the interview data implied that one reason for choosing to build systems from COTS components was related to the perceived cost saving potential of this method. However, with the development and maintenance of COTS-based systems there were challenges with managing cost. The practicalities of the COTS-based approach can result in practitioners having to balance different cost challenges in order to produce systems which meet requirements and to manage their maintenance over the system lifecycle.

It could be seen that one facet of concept *Balancing Cost Challenges* was the cumulative effect of different design principles on cost. Some principles, when considered individually, were considered to contribute to reducing cost, but when combined may conflict and outweigh any individual cost benefits. Furthermore, system developers and maintainers may be forced to select inappropriate products, processes or principles as a result of other pressures in order to deliver a functioning system. Therefore, system developers and maintainers were balancing the cost reducing choices of selecting certain approaches, products, design principles etc. with the cost increasing effects of other decisions.

Another contributing factor to concept *Balancing Cost Challenges* concerned the assessment of component suitability, which encompassed tasks requiring human effort and skill, thus, incurring financial cost. Thus, *Assessing component suitability PMiA11* was seen to be a contributing factor to *Increasing system development costs PMiA11*, which was related to the amount of effort and skill required to perform these tasks. From the interview data *Balancing Cost Challenges* occurred because one aim of practitioners was to lessen the effect of *Increasing system development costs PMiA11*, which required balancing the perceived cost saving opportunities of using COTS components with the additional costs incurred when assessing and testing the suitability of the components for use within a system.

Concept *Balancing Cost Challenges* was also considered to link codes *Integration effort PMiA8* and *Increasing system development costs PMiA11*. *Integration effort PMiA8* related to the human effort incurred when integrating COTS-components and other system parts. *Integration effort PMiA8* was seen to be a contributing factor to *Increasing system development costs PMiA11* because human effort incurs financial cost. Again, the aim of practitioners was to make design decisions to produce systems
which supported the system requirements, but which also minimised *Integration effort PMiA8*; decisions requiring them to balance the different cost challenges.

The constant comparative process also identified the following combination of codes to be linked to concept *Balancing Cost Challenges*:

- **System developers reusing COTS components PMiA1** and **Maintaining multiple interfaces PMiA3**;
- **Saving development time PMiA2** and **Maintaining multiple interfaces PMiA3**;
- **Saving development time PMiA2** and **Increasing system maintenance effort PMiA3**; **Saving development time PMiA2** and **Conflicting maintenance schedules PMiA10**;
- **Saving development time PMiA2** and **Assessing component suitability PMiA11**;
- **Maintaining multiple interfaces PMiA3** and **Selecting architecturally compatible components PMiA6**;
- **Maintaining multiple interfaces PMiA3** and **Designers minimising system change PMiA15**.

Further concepts, which emerged from other combinations of the codes, are presented in Table 5.2.

<table>
<thead>
<tr>
<th>Concepts</th>
<th>Key Text Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost Reducing Strategy</td>
<td>PMiA1, PMiA2, PMiA3 PMiA4, PMiA5, PMiA7 PMiA10</td>
</tr>
<tr>
<td>Reducing integration effort</td>
<td>PMiA2, PMiA8</td>
</tr>
<tr>
<td>Balancing Cost Challenges</td>
<td>PMiA2, PMiA3, PMiA6 PMiA7, PMiA8, PMiA10, PMiA11, PMiA15</td>
</tr>
<tr>
<td>Vendor Homogeneity</td>
<td>PMiA2, PMiA9</td>
</tr>
<tr>
<td>Appreciating Cultural Factors</td>
<td>PMiA2, PMiA13</td>
</tr>
<tr>
<td>Reducing System Complexity</td>
<td>PMiA2, PMiA4, PMiA14</td>
</tr>
<tr>
<td>Design Objective</td>
<td>PMiA2, PMiA14</td>
</tr>
<tr>
<td>Lacking Common Understanding</td>
<td>PMiA2, PMiA16, PMiA17</td>
</tr>
<tr>
<td>Design Decision</td>
<td>PMiA2, PMiA18, PMiA19, PMiA14</td>
</tr>
<tr>
<td>Conflicting Design Decisions</td>
<td>PMiA2, PMiA18, PMiA19</td>
</tr>
<tr>
<td>Increasing System Complexity</td>
<td>PMiA3</td>
</tr>
<tr>
<td>Conflicting Design Principles</td>
<td>PMiA3, PMiA4, PMiA5</td>
</tr>
<tr>
<td>Offsetting Cost Challenges</td>
<td>PMiA3, PMiA6</td>
</tr>
<tr>
<td>Offsetting Maintenance Cost</td>
<td>PMiA3, PMiA7, PMiA15</td>
</tr>
<tr>
<td>Managing Architectural Complexity</td>
<td>PMiA3, PMiA8</td>
</tr>
<tr>
<td>System Maintenance Complexity</td>
<td>PMiA3, PMiA10</td>
</tr>
</tbody>
</table>
The constant comparative method (Glaser, 1978) was used to compare each concept with all other concepts, grouping them into Categories which shared even broader commonalities. The aim of this stage of the analysis was to define initial categories, which emerged from concepts, which in turn emerged from the first interview. These categories served as a focus for subsequent interview sessions. However, it should be noted that the categories and links between concepts were later refined following the emergence of additional concepts originating from the analysis of further interviews.

The following concepts were grouped into the following categories:

*Reducing Integration Effort; Cost Reducing Strategy and Balancing Cost Challenges* shared the common theme of **COST ISSUES**. The reasons for these groupings are as follows: It was seen that factors contributing to *Reducing Integration Effort* related to cost because effort affected cost. *Cost Reducing Strategy* and *Balancing Cost Challenges* were also considered to influence cost.

Category **COMPLEXITY ISSUES** emerged from the following concepts:

- *System maintenance complexity; Increasing System Complexity and Managing Architectural Complexity* shared the common theme of ‘Complexity’.

Category **DESIGN ISSUES** emerged from grouping the following concepts because they all referenced *design*:

- *Managing Architectural Complexity; Conflicting Design Principles; Balancing Design Principles; Design Objective; Conflicting Design Decisions; Design Decision; Designing For Change and Vendor Homogeneity*. 

**Table 5.2: Emergence of concepts from open codes in Interview A.**

<table>
<thead>
<tr>
<th>Concepts</th>
<th>Key Text Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Design Principle</td>
<td>PMiA2, PMiA3, PMiA15</td>
</tr>
<tr>
<td>Architects Balancing Design Principles</td>
<td>PMiA5, PMiA6</td>
</tr>
<tr>
<td>Architects Compromising Design Principles</td>
<td>PMiA5, PMiA9</td>
</tr>
<tr>
<td>Increasing Maintenance Complexity</td>
<td>PMiA3, PMiA19, PMiA10</td>
</tr>
<tr>
<td>Cultural Misunderstandings</td>
<td>PMiA9, PMiA10, PMiA13</td>
</tr>
<tr>
<td>Designing For Change</td>
<td>PMiA14, PMiA15</td>
</tr>
</tbody>
</table>
Category **CULTURAL ISSUES** emerged from the following concepts because they were considered to relate to a cultural aspect from the data:

- *Lacking Common Understanding; Appreciating Cultural Factors and Cultural Misunderstandings.*

At this stage of the analysis category **COST ISSUES** was identified as a candidate for the *Core category* because although groupings of other concepts led to the emergence of many categories they all appeared to be linked in some way to **COST**. Glaser (1978, p. 94) defined the core category as relating to other categories and as “the main theme, for what is the main concern or problem for the people in the setting”. For example, although category **DESIGN ISSUES** encompassed concepts relating to system design it also appeared that these issues were influenced by **COST**. For example, concept *Designing For Change* was linked to category **DESIGN ISSUES** by the common theme of ‘design’. However, concept *Designing For Change* emerged from the data as a prescription for reducing the ongoing costs of maintaining COTS-based systems. This was justified by the proposition from the interview data that systems tend to change over time. Thus, it was felt that designing COTS-based systems to be changed would contribute to reducing ongoing maintenance costs.

The categories and associated concepts, which emerged from Interview A, are displayed in Table 5.3.

<table>
<thead>
<tr>
<th>Category</th>
<th>Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>COST ISSUES</strong></td>
<td>Reducing Integration Effort</td>
</tr>
<tr>
<td></td>
<td>Strategy For Reducing Costs</td>
</tr>
<tr>
<td></td>
<td>Balancing Cost Challenges</td>
</tr>
<tr>
<td><strong>COMPLEXITY ISSUES</strong></td>
<td>Increasing Maintenance Complexity</td>
</tr>
<tr>
<td></td>
<td>System Maintenance Complexity</td>
</tr>
<tr>
<td></td>
<td>Increasing System Complexity</td>
</tr>
<tr>
<td></td>
<td>Managing Architectural Complexity</td>
</tr>
<tr>
<td><strong>DESIGN ISSUES</strong></td>
<td>Managing Architectural Complexity</td>
</tr>
<tr>
<td></td>
<td>Conflicting Design Principles</td>
</tr>
<tr>
<td></td>
<td>Balancing Design Principles</td>
</tr>
<tr>
<td></td>
<td>Design Objective</td>
</tr>
<tr>
<td></td>
<td>Conflicting Design Decisions</td>
</tr>
<tr>
<td></td>
<td>Design Decision</td>
</tr>
<tr>
<td></td>
<td>Designing For Change</td>
</tr>
<tr>
<td></td>
<td>Vendor Homogeneity</td>
</tr>
<tr>
<td><strong>CULTURAL ISSUES</strong></td>
<td>Lacking Common Understanding</td>
</tr>
<tr>
<td></td>
<td>Appreciating Cultural Factors</td>
</tr>
</tbody>
</table>
A diagrammatical representation of the concepts and categories which emerged from Interview A is presented in Figure 5.2.

**Figure 5.2: Cluster diagram providing diagrammatical representation of concepts and categories emerging from Interview A.**

The purpose of the cluster diagram displayed in Figure 5.2 was to present a visual representation of the concepts and categories which emerged from Interview A. In the diagram the categories are displayed in upper case. The concepts are arranged around the outer part of the diagram. The key text identifiers associated with each concept are displayed. The arrows indicate the direction of the relationships. For example, concepts *Designing For Change* and *Vendor Homogeneity* were associated with the broader concept of *Design Decision*, which in turn was associated with the broader category of *DESIGN ISSUES*.

### 5.7 Summary of Analysis from Remaining Interviews

The aim of this section is to provide a summary of the results of the analysis of the remaining interviews. Further details of this analysis are presented in Appendices B and C.
Chamaz (2006) proposed that once an analytical direction had been established from open coding focused coding should be used. Glaser (1978, p. 61) termed this as *Selective coding*. Selective coding was used to synthesise and explain larger segments of data. It allowed for larger chunks of data to be sifted through. This approach was used to identify and assign codes to key text from the remaining interviews. The constant comparative method was used to recognise commonalities of these codes with concepts which emerged from Interview A and, where relevant, to identify additional emerging concepts. The list of concepts, including key text identifiers, which emerged from all eleven interviews, is presented in Table 5.4.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Key Text Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost Reducing Strategy</td>
<td>PMiA1, PMiA2, PMiA3, PMiA4, PMiA5, PMiA7, PMiA10, ARiB9, ARiB10, ARiB13, ARiC12, ARiC15, ARiC17, ARiD1, ARiD9, ARiD23, ARiD24, ARiF1, ARiF2, ARiL12</td>
</tr>
<tr>
<td>Reducing User Intervention</td>
<td>ARiB9, ARiB10, ARiC15, ARiC17</td>
</tr>
<tr>
<td>Implications Of Change</td>
<td>ARiC2, ARiC5</td>
</tr>
<tr>
<td>Establishing Relationships</td>
<td>ARiD23, ARiD25, ARiD26, ARiD28, ARiL18, ARiL19, ARiL20</td>
</tr>
<tr>
<td>Reducing Integration Effort</td>
<td>PMiA2, PMiA8, ARiD1, ARiD2, ARiD3, ARiD4, ARiF1, ARiF2</td>
</tr>
<tr>
<td>Reducing Potential Problems</td>
<td>PMiH6</td>
</tr>
<tr>
<td>Balancing Cost Challenges</td>
<td>PMiA2, PMiA3, PMiA6, PMiA7, PMiA8, PMiA10, PMiA11, PMiA15, ARiB3, ARiB8, ARiB9, ARiB10, ARiB11, ARiB12, ARiB18, ARiC15, ARiC20, ARiD34, SDiE1, SDiE2, SDiE4, SDiE5, SDiE6, SDiE7, ARiK3, ARiK4, ARiL12</td>
</tr>
<tr>
<td>Knock-On-Effect</td>
<td>ARiC2, ARiC10, SDiE11, SDiE12, SDiE13</td>
</tr>
<tr>
<td>Increasing Cost</td>
<td>ARiB3, ARiC2, ARiC3, PMiH7, ARiK5</td>
</tr>
<tr>
<td>Component Licensing Fees</td>
<td>ARiB10, ARiD34</td>
</tr>
<tr>
<td>Staffing Cost</td>
<td>ARiL5, ARiL6</td>
</tr>
<tr>
<td>Increasing Effort</td>
<td>ARiB2, ARiB3, ARiB6, ARiB7, ARiD14, ARiD15, SDiE1, SDiE4, SDiE8, PMiG1, PMiG2, PMiG3, PMiG4, PMiG5, PMiG6, ARiL5, ARiL11</td>
</tr>
<tr>
<td>Concept</td>
<td>Key Text Identifier</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>Change Unpredictability</td>
<td>ARiC2, ARiC4, ARiC7</td>
</tr>
<tr>
<td>Managing Change</td>
<td>- PMiH2, PMiH1, PMiH7, ARiK8</td>
</tr>
<tr>
<td>Redoing Integration Work</td>
<td>ARiC2, ARiC9, PMiG1, PMiG2, PMiG3, PMiG4, PMiG5, PMiG6</td>
</tr>
<tr>
<td>Beyond Sphere Of Influence</td>
<td>ARiB23, ARiB24, ARiC16, ARiC21, ARiD14, ARiD15, ARiD16, ARiD21, ARiK6, ARiL4</td>
</tr>
<tr>
<td>Organisational Concerns</td>
<td>ARiB2, ARiB15, ARiB16, ARiB17</td>
</tr>
<tr>
<td>Support Quality</td>
<td>ARiK9, ARiK10, ARiK11, ARiL16</td>
</tr>
<tr>
<td>Losing Faith</td>
<td>ARiB2, ARiD14, ARiD17, ARiD18</td>
</tr>
<tr>
<td>Denying Responsibility</td>
<td>ARiD16, ARiD17, ARiD18, ARiD19, SDiE1, SDiE4, SDiE10</td>
</tr>
<tr>
<td>Resisting Change</td>
<td>PMiH11, PMiH13, PMiH16</td>
</tr>
<tr>
<td>Conflicting Business Motives</td>
<td>ARiC13, ARiC14, ARiK6</td>
</tr>
<tr>
<td>Lacking Common Understanding</td>
<td>PMiA2, PMiA16, PMiA17, PMiJ9, PMiJ10</td>
</tr>
<tr>
<td>Appreciating Cultural Factors</td>
<td>PMiA2, PMiA13</td>
</tr>
<tr>
<td>Cultural Misunderstandings</td>
<td>PMiA9, PMiA10, PMiA13</td>
</tr>
<tr>
<td>Design Decision</td>
<td>PMiA2, PMiA3, PMiA18, PMiA19, PMiA14, PMiA15, ARiB10, ARiB12, ARiB13, ARiC15, ARiC16, PMiG12, ARiK3, ARiK4, ARiK9</td>
</tr>
<tr>
<td>Designing For Change</td>
<td>PMiA14, PMiA15, ARiC2, ARiC7</td>
</tr>
<tr>
<td>Vendor Homogeneity</td>
<td>PMiA2, PMiA9, ARiB14, ARiB15, PMiG12</td>
</tr>
<tr>
<td>Conflicting Design Decisions</td>
<td>PMiA2, PMiA18, PMiA19, ARiB2, ARiB4, ARiB14, ARiB13, SDiE1, SDiE4, SDiE8, ARiF4, ARiF12</td>
</tr>
<tr>
<td>Design Objective</td>
<td>PMiA2, PMiA14, ARiD23, ARiD24, ARiF1, ARiF2, ARiF1, ARiF3</td>
</tr>
<tr>
<td>Balancing Design Principles</td>
<td>PMiA5, PMiA6, PMiA9, ARiC15, ARiC18, ARiF4, ARiF11, ARiF12, PMiG11</td>
</tr>
<tr>
<td>Conflicting Design Principles</td>
<td>PMiA3, PMiA4, PMiA5, ARiC15, ARiC19, ARiF4, ARiF12</td>
</tr>
<tr>
<td>Increasing Maintenance Complexity</td>
<td>PMiA3, PMiA10, PMiA19, ARiF3, ARiF4, ARiF7, ARiF8, PMiG1, PMiG2, PMiG3, PMiG4, PMiG5, PMiG6</td>
</tr>
<tr>
<td>System Maintenance Complexity</td>
<td>PMiA3, PMiA10, ARiD25</td>
</tr>
<tr>
<td>Reducing System Complexity</td>
<td>ARiB3, ARiC15, ARiC16, ARiD1, ARiD9</td>
</tr>
<tr>
<td>Increasing System Complexity</td>
<td>PMiA3</td>
</tr>
<tr>
<td>Degree Of Dependency</td>
<td>ARiC1, ARiC2, ARiC3, ARiF4,</td>
</tr>
</tbody>
</table>
Table 5.4: List of concepts which emerged from all interviews, including key text identifiers.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Key Text Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributing Accountability</td>
<td>SDiE1, SDiE4, SDiE9</td>
</tr>
<tr>
<td>System Complexity</td>
<td>ARiL22</td>
</tr>
<tr>
<td>Component Complexity</td>
<td>ARiL21</td>
</tr>
<tr>
<td>Increasing Support Complexity</td>
<td>SDiE1, SDiE4, SDiE5, SDiE6, SDiE7, ARiK9, ARiK10, ARiK11</td>
</tr>
</tbody>
</table>

The full list of categories and associated concepts, which emerged from the analysis of the interviews, is presented in Table 5.5.

<table>
<thead>
<tr>
<th>Category</th>
<th>Concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTROLLING COST</td>
<td>Cost Reducing Strategy</td>
</tr>
<tr>
<td></td>
<td>Reducing Integration Effort</td>
</tr>
<tr>
<td></td>
<td>Reducing Human Intervention</td>
</tr>
<tr>
<td></td>
<td>Reducing Potential Problems</td>
</tr>
<tr>
<td></td>
<td>Establishing Effective Relationships</td>
</tr>
<tr>
<td></td>
<td>Balancing Cost Challenges</td>
</tr>
<tr>
<td></td>
<td>Increasing Cost</td>
</tr>
<tr>
<td></td>
<td>Increasing Effort</td>
</tr>
<tr>
<td></td>
<td>Change Unpredictability</td>
</tr>
<tr>
<td></td>
<td>Redoing Integration Work</td>
</tr>
<tr>
<td></td>
<td>Implications Of Change</td>
</tr>
<tr>
<td></td>
<td>Knock-On-Effect</td>
</tr>
<tr>
<td></td>
<td>Component Licensing Fees</td>
</tr>
<tr>
<td></td>
<td>Staffing Cost</td>
</tr>
<tr>
<td>ORGANISATIONAL ISSUES</td>
<td>Beyond Sphere Of Influence</td>
</tr>
<tr>
<td></td>
<td>Organisational Concern</td>
</tr>
<tr>
<td></td>
<td>Support Quality</td>
</tr>
<tr>
<td></td>
<td>Losing Faith</td>
</tr>
<tr>
<td></td>
<td>Denying Responsibility</td>
</tr>
<tr>
<td></td>
<td>Conflicting Business Motives</td>
</tr>
<tr>
<td></td>
<td>Resisting Change</td>
</tr>
<tr>
<td>DESIGN PRINCIPLES</td>
<td>Design Decision</td>
</tr>
<tr>
<td></td>
<td>Conflicting Design Decisions</td>
</tr>
<tr>
<td></td>
<td>Vendor Homogeneity</td>
</tr>
<tr>
<td></td>
<td>Designing For Change</td>
</tr>
<tr>
<td></td>
<td>Design Objective</td>
</tr>
<tr>
<td></td>
<td>Balancing Design Principles</td>
</tr>
<tr>
<td></td>
<td>Conflicting Design Principles</td>
</tr>
<tr>
<td>MANAGING COMPLEXITY</td>
<td>Maintenance Complexity</td>
</tr>
<tr>
<td></td>
<td>System Complexity</td>
</tr>
<tr>
<td></td>
<td>Degree Of Dependency</td>
</tr>
<tr>
<td></td>
<td>Relationship Complexity</td>
</tr>
<tr>
<td></td>
<td>Attributing Accountability</td>
</tr>
<tr>
<td></td>
<td>Denying Responsibility</td>
</tr>
<tr>
<td>MANAGING CHANGE</td>
<td>Managing Change</td>
</tr>
<tr>
<td>Category</td>
<td>Concept</td>
</tr>
<tr>
<td>-------------------------</td>
<td>--------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>Designing For Change</td>
</tr>
<tr>
<td></td>
<td>Implications Of Change</td>
</tr>
<tr>
<td></td>
<td>Change Unpredictability</td>
</tr>
<tr>
<td></td>
<td>Knock-On-Effect</td>
</tr>
<tr>
<td></td>
<td>Resisting Change</td>
</tr>
<tr>
<td>CULTURAL ISSUES</td>
<td>Cultural Misunderstanding</td>
</tr>
<tr>
<td></td>
<td>Appreciating Cultural Factors</td>
</tr>
<tr>
<td></td>
<td>Lacking Common Understanding</td>
</tr>
</tbody>
</table>

Table 5.5: List of categories and associated concepts, which emerged from all of the interviews.

The following sections examine the relationships between the concepts and categories.

5.8 Theoretical Sorting and Coding: Analysis of Relationships between Concepts and Categories

The aim of this part of the analysis was to examine the relationships between concepts to uncover new connections and patterns between concepts in order refine existing categories or uncover new categories. The methods used to perform these tasks were Theoretical Sorting and Theoretical Coding (Glaser, 1978; 1998).

Theoretical Sorting is the conceptual sorting of ideas, not data, to determine how concepts relate to other ideas to form categories to be integrated into a theory (Glaser, 1978; 1998; Charmaz, 2006). Theoretical Sorting was performed by arranging the memos over a large space (in the case of this study a large kitchen table was used) and arranging them into groups displaying similar properties, connections, underlying processes and conceptual orderings.

The purpose of Theoretical Coding was to define how the concepts and categories related to each. Glaser (1978; 1998) claimed that the explanation of the relationships between concepts and categories should form the basis of a Grounded Theory. Glaser (1978; 1998) developed Theoretical Codes to explain the relationships between concepts, and between categories. He maintained that the complexities, which existed between social processes, could not be logically determined or modelled. Thus, it was from the process of Theoretical Coding that a Grounded Theory would be generated to explain what was going on.
From Interview A several candidate categories emerged. However, additional concepts emerged from the analysis of further interviews which required the modification of some initial category labels.

The following diagram, Figure 5.3, is a cluster diagram, created following Theoretical Sorting, which displays an overview of the relationships between the concepts and categories which emerged from the analysis of all eleven interviews. The core category, **CONTROLLING COST** is displayed in uppercase bold and underlined. The other categories are displayed in bold uppercase. The concepts are displayed in italics. The key text identifiers associated with each concept are displayed. The arrows indicate an association between concepts. This diagram is an extension of the cluster diagram presented in Figure 5.2, which was constructed following the analysis of only the first interview.

A detailed explanation of the links between concepts and categories is provided in the following sections. The same naming convention is used in the text: the core category is displayed in uppercase bold and underlined, categories are displayed in bold uppercase, concepts displayed in italics. Open codes are also displayed in italics and include the key text identifier.

The full analysis of the interview data is detailed in Appendices B and C.
Figure 5.3: Cluster diagram providing a visual representation of the relationships between concepts and categories which emerged from the eleven interviews.
5.8.1 Category CONTROLLING COST

As seen in section 5.6.2 concepts Reducing Integration Effort, Cost Reducing Strategy and Balancing Cost Challenges were initially considered to be associated with category COST ISSUES. However, following Theoretical Sorting all of the following concepts also appeared to share a common theme of CONTROLLING COST:

- Cost Reducing Strategy
- Reducing Integration Effort
- Reducing Human Intervention
- Reducing Potential Problems
- Establishing Effective Relationships
- Balancing Cost Challenges
- Increasing Cost
- Increasing Effort
- Change Unpredictability
- Redoing Integration Work
- Implications Of Change
- Resisting Change
- Knock-On-Effect
- Component Licensing Fees
- Staffing Cost

From the interview data it appeared that software practitioners (system developers, maintainers) considered that the relationship between these concepts encompassed more impact than just being ‘cost issues’. From a software practitioners perspective they were resigned to the fact that that developing and maintaining systems incurred cost. In fact, cost appeared to be the main reference point for most decisions. It was clear that without cost being a governing factor any system could be produced, which supported all requirements. However, IBM is a commercial company and most decisions were made in relation to cost, winning business and potential profit. It could be suggested that other commercial companies are controlled by the same pressures. It could also be suggested that public departments are also subject to similar concerns, especially when they have to demonstrate accountability and value for money of public finances. Therefore, many
of the decisions taken or principles supported were in relation to attempts of ‘controlling cost’. For example, the choice to develop software systems using COTS components is normally made for reasons of cost; the development costs are considered to be cheaper compared with developing a system from scratch. However, once the COTS-based development method has been chosen other decisions may have conflicting influences on cost. For example, selecting components supporting the same architectural standard may contribute to reducing costs because they are more likely to integrate with less effort. However, to satisfy customer requirements there is no guarantee that the available components will support the same architectural standard. Therefore, integrating and maintaining components not supporting the same architectural standard can require more effort, thus contributing to increasing costs.

It can be suggested that category **CONTROLLING COST** is not just related to the domain of COTS-based system development but can apply to numerous domains. In many areas of commercial and public life decisions relating to **CONTROLLING COST** are being made.

A diagrammatical representation of the relationship between concepts linking to category **CONTROLLING COST**, highlighted in black, is presented in Figure 5.4.
Figure 5.4 shows that **CONTROLLING COST** is influenced by different factors, such as, *Cost Reducing Strategy*, *Balancing Cost Challenges* and *Increasing Cost*. The diagram also indicates that *Cost Reducing Strategy*, *Balancing Cost Challenges* and *Increasing Cost* are also influenced by other factors.

Details of the relationships between concepts associated to category **CONTROLLING COST**, as displayed in Figure 5.4 is presented below.

**5.8.1.1 Concept Cost Reducing Strategy**

From the data concept *Cost Reducing Strategy* emerged as an underlying process employed (or considered) by software practitioners (IT architects, Project managers etc.) of COTS-based system design and management. The commercial challenges and pressures of the environments in which the practitioners operated forced them to employ *Cost Reducing Strategies*. Reducing costs appeared to be a common goal for all of the interviewees, whether they were IT architects, project managers or software support personnel.

Furthermore, from the data the notion of cost was associated to two phases: the cost to develop systems; and the ongoing cost to maintain systems. Although, different actions were performed during development or maintenance phases of projects by the practitioners the commonality was that they still adopted *Cost Reducing Strategies*. However, some *Cost Reducing Strategies* affecting maintenance cost had to be thought about during the development phase of systems because they involved system design decisions, which would be difficult to implement once system had been implemented.

Concept *Cost Reducing Strategy* comprises of three parts: the concepts of *Cost, Reducing* and *Strategy*.

On *Cost*, although the final measure may have been a financial, monetary value the concept was used interchangeably with other concepts, such as, *Effort, Time* or *Requiring Skill*. The assumption from the data was that these concepts influenced the final monetary value.
**Effort**, for example, related to human effort, which in turn equated to a financial value, such as an employee’s salary. Thus, a consequence of greater numbers of people working on a task was extra cost.

**Time** had similar properties to effort in that the longer people spent working on task the higher the financial cost were due to the additional monetary value spent on salaries. However, saving time could have additional benefits over just the financial savings in wages because, in many instances, delivering a project to an agreed timescale avoided late delivery financial penalty charges.

**Requiring skill** related to the skills that people performing the tasks must possess. Some tasks were more specialised than others, thus required people to hold specialist skills. However, the cost of employing people with specialist skills was normally considered higher than the costs to employ people with general skills. Therefore, employing a person with specialist skills over a longer period of time had a greater effect on cost than employing a person with general skills over the same period of time.

From the data the concept of Reducing indicated one aim of software practitioners which, in this case, was Reducing Cost. However, the aim of Reducing Cost normally required some sort of action which practitioners hoped would result in Reducing Cost – i.e. reducing cost did not just occur in isolation of the decision to perform an action. Thus, to instigate Reducing Cost software practitioners consciously selected strategies (tactics, techniques), which they believed would result in Reducing Cost. Glaser (1978) stated that the defining quality of a strategy is whether it is consciously selected. He explained that if an action is not consciously selected it is merely a consequence of some other action.

Glaser’s (1978, p. 74) ‘Six C’s theoretical coding family, which comprised of the following codes: causes, contexts, contingencies, consequences, covariance’s and conditions, can assist with the explanation of the relationships between the constituent parts of concept Cost Reducing Strategy (cost, reducing and strategy) for the following reasons:

As indicated above, an action is defined as a strategy if consciously selected with the expectation of achieving a specific end goal. In the case of a Cost Reducing Strategy the
intended end goal was *Reducing Cost*. Thus, with reference to Glaser’s ‘Six C’s coding family (1978) *Cost Reducing Strategy* is the *cause* and *Reducing Cost* the intended *consequence*. The success of the *Cost Reducing Strategy* assumed that suitable *conditions* existed (or the absence of conditions resulting in rising costs).

However, as can be seen in the analysis of the data in this study not all conditions could be predicted or controlled. There were cases where software practitioners consciously selected a *Cost Reducing Strategy* believing that all conditions had been accounted for when something untoward occurred, which was not be predicted. The consequence of an unpredicted event can be *Increasing Costs*. For example, one proposed design principle was for IT architects to build COTS-based systems from as few, larger components as possible. This was considered a *strategy* because it arose from the conscious decision related to the belief that integration costs would be reduced as a result of saving effort by integrating fewer components. One assumption was that other conditions, such as the necessity to spend additional effort disabling redundant functionality of larger components, was not required.

**Need for a Cost Reducing Strategy**

From the data it appeared that the requirement for cost reducing strategies arose as a consequence of the competition and commercial pressures experienced by IT companies, such as IBM, to reduce costs.

From a cost perspective there were different phases of activity which had different cost implications. Firstly, during system development phases the cost challenges were to develop systems which adhered to customer requirements, were implemented within agreed time frames and delivered within the estimated costs.

Secondly, once implemented there were cost challenges to keep systems running. A contributing factor to the ongoing costs of COTS-based systems was *Change*. Ongoing costs could arise as a result of *Managing Change* instigated by component vendors, operating system upgrades and patching, hardware changes and changes in system requirements initiated by customers.
When a system is handed over to the customer following implementation then the customer would normally responsible for these costs. However, when IBM manages a system on a customer’s behalf then IBM would be responsible for the ongoing costs, as documented within the contract between the organisations.

Therefore, the implementation of a *Cost Reducing Strategy* was seen as a contributing factor to **CONTROLLING COST** because there were other factors which could lead to *Increasing Costs*. Thus, **CONTROLLING COST** involved managing all factors relating to *Cost*.

### 5.8.1.2 Concept Reducing Integration Effort

*Integration effort* related to the human effort required to integrate COTS components. From the data it was seen that concept *Integration Effort* was a contributing factor of *Cost* because human effort incurs financial cost related to the skill and salary to perform these tasks.

It can be seen that *Integration Effort* was a factor which could vary as a *consequence* of other actions. Thus, the amount of *Integration Effort* was related to the types of tasks performed (actions), the skill required in performing the tasks and the time taken to perform tasks. The following diagram illustrates this:

Integration Action/decision $\rightarrow$ can result in $\rightarrow$ *Increasing or Reducing integration effort*.

However, from the interview data there was no indication of what constituted a benchmark level of *Integration Effort*. For example, if certain decisions were deemed to contribute to *Reducing Integration Effort* the *baseline* amount of ‘integration effort’, to which this was compared, was not established. The data only indicated that there were decisions, actions etc. which could result in *Increasing Integration Effort* and other decisions and actions which could result in *Reducing Integration Effort*.

Therefore, *Integration Effort* could be viewed as a continuum, of which *Reducing Integration Effort* was at one end. The diagram below illustrates this.

Increasing integration effort $\leftrightarrow$ *Integration effort* $\rightarrow$ Reducing integration effort.
Thus, the data indicated that actions, principles and decisions could be endorsed by COTS system developers because they were believed to contribute to Reducing Integration Effort.

With reference to Glaser’s (1978, p. 74) Theoretical Codes it could be seen that when conscious decisions, relating to the design of COTS-based systems, were made by practitioners with the aim of Reducing Integration Effort this could be viewed as a Cost Reducing Strategy. However, when decisions were not consciously made then any ‘reduction’ in Integration Effort should be viewed as a consequence of the action.

5.8.1.3 Concept Reducing Human Intervention

Concept Reducing Human Intervention related to the concept of Human Intervention. There was a relationship between Human Intervention and Effort, and thus, Cost because Human Intervention implied someone performing an action, thus requiring human effort, which normally incurred Cost.

With regard to the development and maintenance of COTS-based systems tasks involving human intervention contribute to cost. Tasks requiring specialist skill normally incur greater cost.

Therefore, actions which resulted in Reducing Human Intervention contributed to reducing effort and thus, reducing cost. With reference to Glaser’s (1978, p. 75) Theoretical Codes Reducing Human Intervention was seen to be an example of a Cost Reducing Strategy when the underpinning decisions were made with the intention of consciously Reducing Human Intervention. However, where this was not the case then Reducing Human Intervention merely become the consequence of an action.

Reducing Human Intervention was also seen to relate to the ‘degree’ coding family (Glaser, 1978, p. 75) because it could be viewed as part of a continuum, ranging from Reducing Human Intervention by a small amount, through to Reducing Human Intervention completely. Thus, Reducing Human Intervention implied that there was the potential for an action to occur without human intervention which would have the biggest influence on reducing cost.
5.8.1.4 Concept Reducing Potential Problems

From the data it was suggested that COTS-based system developers and maintainers understood that some design decisions would increase the chance for potential problems. Resolving problems contributed to Increasing Costs as a result of the additional time, effort and resources required to perform problem determination and resolution tasks. Therefore, a conscious decision aimed at Reducing Potential Problems can be viewed as a Cost Reducing Strategy if made in relation to reducing the costs associated with managing problems. However, choices which merely result in Reducing Potential Problems should be seen as a consequence of that choice, rather than as part of a strategy.

5.8.1.5 Concept Establishing Effective Relationships

From the data Establishing Relationships was proposed as key aim to enable system developers and maintainers to gain cooperation from other parties.

It can be seen that Establishing Relationships can be measured in terms of degree (Glaser, 1978, p. 75). This is explained as follows. From the data the assumption was of Establishing ‘effective’ Relationships, which contributed to reducing costs by Aiding integration ARiL19 and Aiding maintenance ARiL20 activities. However, at the other end of the continuum, the effect of system developers, maintainers, vendors or customers Establishing ‘ineffective’ Relationships was the likelihood that this would contribute to Increasing Cost as a result of integration or maintenance problems not being resolved effectively or in a timely manner.

Furthermore, if system developers, maintainers or customers consciously attempt to establish ‘effective’ relationships with vendors, with the aim of reducing costs, it can be seen to be an example of a Cost Reducing Strategy. It is the conscious act which would define this as a strategy (Glaser, 1978).

5.8.1.6 Concept Balancing Cost Challenges

From the data it was suggested that in an ideal world COTS system developers and maintainers would aim to implement design decisions which contributed to reducing costs. However, in the real world these options were not always available.
For example, building systems with fewer components (Using fewer components ARiB18) was considered to be a contributing factor to reducing total lifecycle costs because it was assumed that less effort would be required to configure and maintain a lesser number of interfaces. Thus, Using fewer components ARiB18 was deemed to be an approach to address the cost challenges of developing and maintaining systems.

However, Using fewer components ARiB18 to build COTS-based systems could result in some components being functionally larger in size, compared with building the same systems from greater numbers of components, where the assumption was that some components would be smaller in size in order to deliver the same amount system functionality. From the interview data it was seen that Using fewer components ARiB18 could result in Functional redundancy ARiB3, where more functionality then was required was provided by larger components. Using components with Functional redundancy ARiB3 could require additional effort to disable redundant functionality (Redundant functionality was normally disabled because leaving it intact requires additional memory and other system processes and resources).

The net result was a requirement for system developers to ‘balance’ the Cost Challenges related to reducing integration and ongoing costs by Using fewer components ARiB18, with the increasing costs associated with disabling Functional redundancy ARiB3.

However, from the data it was not clear if practitioners implemented Balancing Cost Strategies by consciously applying DESIGN PRINCIPLES to balance costs, or whether Balancing Cost Challenges was a consequence of making design decisions or applying design principles. The issue was that during the system development phase system developers and maintainers would not know what the total system lifecycle costs would be. Therefore, Balancing Cost Challenges was more likely related to system developers considering the implementation of design ‘best practices’ which, by their nature, aimed to take into account the notion that the volatility of the COTS market would affect development and maintenance costs. For example, the concept of building systems from components supporting the same architectural standard underlined the assumption that such components would require less effort to maintain during the life of a system than components supporting different architectural standards.
With reference to Glaser’s strategy theoretical coding family (1978, p. 76), it can be seen that the application of a Cost Reducing Strategy was to consciously offset the effect of cost increasing factors, thus contributing to Balancing Costs Challenges. For example, when costs increase as the result of unforeseen circumstances the consequence of Cost Reducing Strategies would aim to reduce costs, thus, contributing to balancing costs.

With reference to the Six C’s coding family (Glaser, 1978, p. 74) it could be seen that the consequence of system developers implementing Design Decisions was to contribute to Balancing the Cost Challenges of developing and maintaining COST-based systems, which in turn contributed to category CONTROLLING COST. This was because CONTROLLING COST can involve implementing varied methods to balance the different Cost Challenges of COTS-based design. There will always be contributing factors to cost. Balancing Cost Challenges involved lessening the impact of these factors on the overall cost.

5.8.1.7 Concept Increasing Cost

From the data concept Increasing Cost encompassed decisions made by system developers which resulted in, or were are believed to contribute to Increasing Cost. This was opposed to other decisions and actions which were believed to result in ‘reducing’ Cost.

With reference to Increasing Cost it could be suggested that the broader concept of Cost could be viewed in terms of degree (Glaser, 1978, p. 75). The data suggested that Cost comprised of a continuum of factors ranging from those causing Increasing Cost through to those resulting in ‘reducing’ Cost. It was also be assumed that some decisions could affect cost differently in terms of degree. For example, one factor may only result in a small degree of Increasing or Reducing Cost, whereas other factors may result in Increasing or Decreasing Cost by a larger amount.

From the data it was seen that system developers employed Cost Reducing Strategies. However, there was no evidence of developers employing Cost ‘Increasing’ Strategies. A proposed reason for this was that for system developers representing commercial organisations the concept of Reducing Cost was a major driving factor because the
primary aim of commercial organisations is to maximise profit; Reducing Cost can contribute to this.

Thus, Reducing Costs could be viewed as an aim of COTS-based system developers whilst Increasing Costs could be seen as an ‘unwelcome’ outcome.

However, the effect of Increasing Cost could be viewed differently by different populations. For example, programmers employed to produce costly component integration code could benefit from additional employment opportunities. Conversely, system developers and maintainers, who were responsible for bearing the cost, viewed the factors contributing to Increasing Cost as just that; factors contributing to increasing cost.

It was seen that the need for system developers to consider factors leading to Balancing Cost Challenges occurred as a consequence of the issues contributing to Increasing Cost. Thus, in an attempt to offset the result of factors causing Increasing Cost system developers and maintainers also considered other factors aimed at ‘Reducing’ Costs. The Implications Of Change and Resisting Change were considered to be contributing factors to Increasing Cost because the actions relating to Change incurred Effort.

The wider category of CONTROLLING COST encompassed concept Increasing Costs because the processes involved with CONTROLLING COST involved other processes relating to Managing Cost, which were Increasing as well as Reducing Cost drivers.

5.8.1.8 Concept Increasing Effort

It was seen that Increasing Effort related to the concept Effort; Increasing Effort reflected that Effort can be measured in terms of degree (Glaser, 1978, p. 75). Thus, there were factors which were considered to contribute to Increasing Effort (of varying degrees) and factors contributing to Reducing Effort (of varying degrees).

However, from the data, Effort was not referenced to in terms of a baseline level of effort. The reference point was in terms of tasks and processes which contributed to Increasing or Reducing Effort, not in terms of a ‘default level’ of effort to which Increasing or Reducing Effort was measured against.
However, it can be suggested that there should be ‘normal levels’ of effort, which apply to different tasks, because tasks, decisions and processes were explained in the data with regard to their consequence on Effort. Therefore, if the perceived consequence of an action is Increasing Effort there must be point of reference level of effort from which increasing effort is assessed.

From the data Increasing Effort was seen as a major contributing factor to Increasing Cost. With COTS-based system development there are different cost contributing factors, such as the costs to purchase hardware and software licenses. However, these costs tended to be predictable and, as such, could be factored in when estimating the costs at the beginning of a project. An issue with the impact of Increasing Effort was that it could not always be fully quantified early on in a project. The nature of the COTS-based development and maintenance process could result in tasks taking longer than anticipated, or unforeseen problems occurring, such as, issues with component integration, issues following component upgrades etc. The consequence of resolving problems was Increasing Effort. For example, on paper the integration of two components may appear straightforward. However, in practice, this integration may end up requiring the production of additional code to facilitate successful integration. Producing additional code contributes to Increasing Effort.

Therefore, as a result of the unpredictable influences of Effort on COTS-based development or maintenance Increasing Effort was perceived as having a greater impact on cost than other cost drivers, such as the cost to purchase or license COTS components. Thus, due to the unpredictable nature of factors requiring ‘effort’ Increasing Effort had a greater impact on system developers and maintainers’ ability to CONTROL COST.

Furthermore, in order for commercial companies to win commissions to develop systems in they normally had to have submit bids. Bids needed to reflect realistic cost estimations in order to realise sufficient profit margins, cover costs, as well as being pitched at a level to be favoured over other competing companies. In cases where the profit margin was small the unpredictability of Increasing Effort challenged the profitability of a winning bid.
5.8.1.9 Concept Redoing Integration Work

Concept Redoing Integration Work is a property of concept ‘Redoing’ – in this case ‘redoing’ integration work. Redoing implies that an action or task has already been performed, thus, resulting in it being performed again. Therefore, when a task or action has incurred Effort it can be seen that Redoing the task contributes to Increasing Effort.

From the data, integration work equated to Integration Effort.

Furthermore, from the data Redoing Integration Work was seen to be a consequence of Change, where change activity can ‘overwrite’ previously performed integration work, necessitating the consequence of Redoing Integration Work.

Redoing Integration Work relates to the category of CONTROLLING COST because it can be seen to be a contributing factor of Increasing Effort.

5.8.1.10 Concept Implications of change

From the data Implications Of Change was associated with negative connotations for system developers and maintainers because Change implied disruption, additional planning, testing, effort and cost. Furthermore, a consequence to changing one system part could be the requirement to change other system parts as a result of underlying dependencies.

Therefore, with reference to Glaser’s (1978, p. 74) Six C’s theoretical coding family a consequence arising from the Implications Of Change can be Increasing Cost.

However, with reference to Glaser’s (1978, p. 74) Six C’s and degree theoretical coding families, it can be seen that the Implications Of Change can be measured in terms of context and degree. For system developers and maintainers the process of Change contributed to Increasing Cost. However, the end result of change activity can be perceived as beneficial to system users. For example, the resolution of software bugs or provision of additional functionality can occur as the result of change activity, being supplied in upgraded or patched components. New system modules created to support new business opportunities are supplied as a result of change activity.
It can also be seen that although change activity can contribute to Increasing Cost for one party, such as a system owner, it may result in additional work opportunities and revenue for other parties, such as system management companies, programmers, IT architects and other IT professionals involved with planning and managing change activity. Component vendors can also benefit from change activity as a result of charging for supplying upgraded or patched components.

From the data, the black-box nature of COTS components does not render them immune from the effect of change. The black box concept implied that whatever occurred to a component’s inner workings the effect on the interfaces should remain as described by the vendor’s documentation. However, from the data, a common consequence of applying patched or upgraded components was of interfaces behaving differently from before (even when vendors claimed that interfaces had not changed).

If the premise is that the Implications Of Change can contribute to Increasing Cost it can be seen that understanding the Implications Of Change enables system developers the opportunity to plan for change activity, limiting the Implications Of Change being a surprise. Thus, understanding the Implications Of Change can contribute to Reducing Cost. Thus, with reference to Glaser’s (1978, p. 76) Strategy theoretical coding family, conscious decisions made in relation to understanding the Implications Of Change, aimed at Reducing the cost of change, can be seen as Cost Reducing Strategies.

5.8.1.11 Concept Knock-On-Effect

Knock-On-Effect related to the consequence of one action on another. For example, a consequence of changing one system part could be the requirement to also change other system parts as a result of dependencies. From the data, Knock On Effect had negative connotations for system developers because it implied ‘additional’ Effort.

Knock-On-Effect can to contribute to Increasing Cost as the resulting actions and effort tend to be unplanned and thus, not factored into original cost estimations.

5.8.1.12 Concept Component Licensing Fees

Component Licensing Fees related to more than just the monetary cost of COTS components. Component Licensing Fees encompassed the intellectual property value of
COTS components to vendors. From a vendor’s perspective Component Licensing Fees were the means by which they aim to recover costs and to make a profit. Thus, commercial decisions by vendors to develop components in the first place were related to market trends and their perceived ability to sell the components.

Conversely, for system developers, Component Licensing Fees were seen to represent predictable costs incurred in lieu of the costs to develop functionality themselves. Thus, for system developers to realise cost savings Component Licensing Fees should be less than the costs to develop and maintain the functionality supplied by components themselves.

Thus, it can be seen that Component Licensing Fees contributed to CONTROLLING COST for two reasons:

When a conscious design decision to use COTS components was made it can be viewed as a Cost Reducing Strategy if the assumption of paying Component Licensing Fees was deemed to outweigh the costs of developing the functionality in-house. Furthermore, the use of COTS components, and the associated Component Licensing Fees, could enable system developers to deliver systems to market faster as a result of saving the time which would be required to produce the functionality in-house. Reducing the time in which systems are brought to market can contribute to Reducing Cost.

Component Licensing Fees contributed to Increasing Cost for system developers as this represented cost expenditure. The contribution of Component Licensing Fees to Increasing Cost may differ between vendors who may charge different amounts for their components. Additionally, Component Licensing Fees tended not to be a one-time cost but normally incurred a cost throughout the life of a component. Furthermore, there was nothing to stop vendors from increasing licensing costs at any time. Thus, system developers may be presented with additional costs leaving them with the dilemma of continuing to pay the licensing costs or bear additional costs in identifying or producing replacement functionality.
5.8.1.13 Concept Staffing Cost

Staffing Cost encompassed the cost of employing people. In the commercial world employing people incurred cost (in other domains, such as the charity sector, some people may not charge for their time). Furthermore, the Staffing Cost varied as the costs of employing people with specialist skills tended to be higher than the costs for those with general skills.

Therefore, with reference to Glaser’s (1978, p. 74) Six C’s and Degree theoretical coding families it can be seen that Staffing Cost was a contributing factor to Increasing Costs. This relationship can be defined in terms of degree, in that the higher the Staffing Cost the greater the contribution to Increasing Cost.

5.8.2 Category ORGANISATIONAL ISSUES

ORGANISATIONAL ISSUES categorised concepts relating to organisations. The way organisations operated and the relationship between organisations differed. Thus, with reference to COTS-based system design the nature of component vendor organisations, their customers’ organisations and the relationships between organisations were related to ORGANISATIONAL ISSUES.

From theoretical sorting the following concepts appeared to share the common theme of ORGANIZATIONAL ISSUES:

- Beyond sphere of influence
- Organisational concerns
- Support quality
- Losing faith
- Denying responsibility
- Conflicting business motives
- Resisting change

A diagrammatical representation of the relationship between concepts linking to category ORGANIZATIONAL ISSUES, highlighted in black, is presented in Figure 5.5.
Figure 5.5: Diagrammatical representation of the relationship between concepts linking to category ORGANISATIONAL ISSUES.

The relationship between the concepts represented in Figure 5.5 is presented in the following sections.

**5.8.2.1 Concept Beyond Sphere Of Influence**

*Beyond Sphere Of Influence* expressed factors which were beyond the control of one party. It also encapsulated different themes: Firstly, the nature of relationships between organisations. The ability of one organisation to acquire specific levels of assistance and cooperation from other organisations may be *Beyond Their Sphere Of Influence* because of differences in the culture of organisations and commercial differences. For example, some organisations have a culture of assisting other parties; whereas others may not provide similar levels of assistance unless specified in a commercial contract.

Secondly, variation to the maintenance schedule or lifespan of components may be *Beyond The Sphere Of Influence* of system developers and maintainers because these decisions tended to be controlled by vendors. For example, vendors tended to impose their own product upgrade cycles on customers, or even decide to withdraw a
component independently of the wishes of customers. Furthermore, a component vendor may cease trading, which is also Beyond The Sphere Of Influence of their customers.

From the data a contributing factor to the success of COTS-based systems was related to the degree (Glaser, 1978) of cooperation existing between organisations. With COTS-based design system developers were reliant upon components which were developed, supplied and maintained by different organisations. Thus, Vendor commitment ARiB23 to providing product support (resolving bugs, assisting with integration issues, etc.) was a crucial characteristic of the relationship, which enabled system developers and maintainers to receive adequate Vendor support quality ARiB24. However, the influence on Vendor commitment ARiB23 and Vendor support quality ARiB24 tended be Beyond [the] Sphere Of Influence for system developers when vendors failed to commit to working closely with them. Thus, because vendor tended not to be part of the same organisation, system developers were not in a position to force a vendor to provide a desired level of service.

Similar scenarios existed within companies. For example, in large organisations internal support teams could behave as if they were from different organisations to those from outside of the teams. There could be a lack of synergy and cooperation between internal teams.

Beyond Sphere Of Influence can be seen to be related to category ORGANISATIONAL ISSUES because [from the data] this concept emerged as a result of organisational differences. It was assumed that people belonging to the same organisation or team would cooperate when all members shared the same collective values of the organisation or team. However, the same collective synergy and level of cooperation tended not to exist between organisations. Furthermore, with separate organisations (or teams and departments which behave like different organisations) the ability of one organisation to influence another may not exist as a result of different business motivations of each organisation.

5.8.2.2 Concept Organisational Concerns

Organisational Concerns encapsulated the confidence and concern one party had over another organisation’s ability to provide a service and had negative connotations as
‘concerns’ suggested dissatisfaction. *Organisational Concerns* could be defined in terms of *degree* (Glaser, 1978, p. 75). Thus, the lower the confidence levels one party has of an organisation the higher their *degree* of *Organisational Concerns*.

It was also seen that the *degree* of *Organisational Concerns* held by party was related to the degree in which *issues of concern* were Beyond [their] *Sphere Of Influence*. Thus it was likely that one party would express a *higher degree* of *Organisational Concern* when the *issues of concern* were completely Beyond [their] *Sphere Of Influence*. For example, in cases where vendors provided poor quality components, were unwilling to cooperate with system developers to improve product quality and where the choice of components were *Beyond The Sphere Of Influence* of system developers because no alternative options were available it was considered that a system developer’s *Organisational Concerns* of a vendor’s organisation would be higher than if he/she had the option to source alternative components from different vendors.

**5.8.2.3 Concept Support Quality**

*Support Quality* conceptualised the quality of the support of COTS components provided by one organisation to another. *Support Quality* could be defined in terms of a *continuum* (Glaser, 1978, p. 75), ranging from values, such as *poor*, through to, *excellent Support Quality*.

From the data *Support Quality* was defined in conjunction with concept *Support Complexity*. From a support perspective, as the number of components supplied by different vendors increased *Support Quality* was seen to decrease, which also resulted in increasing *Support Complexity*. *Support Quality* was seen to increase as a result of the requirement to contact greater numbers of support teams when requiring support. Thus, *Support Complexity* was associated to the number of different support teams.

The *Support Quality* provided by some vendors’ support teams appeared higher when considered in isolation. However, system developers and maintainers’ perceptions of having to deal with increasing numbers of support teams, as the number of components being supplied by different vendors increased, contributed to a feeling of diminishing *Support Quality* (a single point of contact for gaining support for products supplied from multiple vendors could enhance support quality).
Furthermore, with reference to Glaser’s (1978, p. 75) theoretical coding families *Support Quality* and *Support Complexity* could be defined in terms of ‘degree’, measured by the size of a vendor’s organisation. For example, system developers reported that that when acquiring components from one vendor they were more likely to deal with one support team. At the ‘smaller vendor size’ end of the continuum this was considered be true as one team may support all of the vendor’s products. In this case *Support Quality* tended to be higher because one support team provided assistance for all the vendor’s products and could assist with integration issues (however, it should be noted that a single support team could still provide poor quality service). *Support Complexity* could be seen to be low when system developers were required to contact only one support team for all product enquiries.

However, at the other end of the ‘vendor size’ continuum, as vendor organisations became larger it was usual for each product to be supported by a different support team or department within the same organisation. In such cases *Support Quality* tended to decrease because each support team or department could appear as separate organisations to system developers, when little synergy existed between each team or department. In such cases, *Support Complexity* increased as a consequence of the need to contact more than one support team or department in order to receive support for different components supplied by the same vendor.

It was seen that the perceived level of *Support Quality* one party provided to others influenced their perceived levels of *Organisational Concern*. Thus, *poor Support Quality* contributed to ‘higher’ degrees of *Organisational Concern* of the organisation delivering poor quality support. Furthermore, *Support Quality* also tended be *Beyond The Sphere Of Influence* of one organisation when the support service was provided by another organisation.

### 5.8.2.4 Concept Losing Faith

From the data concept Losing Faith encompassed the lack of confidence system developers and maintainers felt as a result of vendors producing poor quality products and in the inability of support organisations to provide adequate product support. However, it can be seen that *Losing Faith* was not limited to organisations. Losing Faith
could occur between individuals. However, the common theme was that *Losing Faith* occurred as a result of inadequate performance. The *degree* of *Losing Faith* was related to the *degree* of *Inadequacy Of Performance*. Thus, higher degrees of *Inadequacy Of Performance* provided one party resulted in greater feelings of *Losing Faith* by other parties.

*Losing Faith* contributed to **ORGANISATIONAL ISSUES** as it related to the inability of an *organisation* to provide quality service.

**5.8.2.5 Concepts Denying Responsibility and Attributing Accountability**

*Denying Responsibility* encapsulated the unwillingness of some organisations to take responsibility for issues relating to their products. This concept could be defined in terms of *degree* because some organisations exploited any opportunity to *Deny Responsibility*, whereas other organisations only adopted this stance when there was supporting evidence.

It can be seen that there were situations where *Attributing Accountability* was inversely related to *Denying Responsibility*. In situations where *Attributing Accountability* was obvious there was less chance of organisations *Denying Responsibility*. However, where *Attributing Accountability* was not clear *Denying Responsibility* by some organisations was more likely to occur.

In complex environments, performing root cause analysis of problem was challenging. This was further compounded when vendors were *Denying Responsibility* for the source of problems when they considered that the problem cause was beyond the scope of their code.

*Attributing Accountability* of problems in complex systems, comprising of more components, greater numbers of interfaces, glue code, wrapper code etc. were considered to be more challenging because vendors would be unable to locate and isolate the root cause of a problem. Furthermore, some vendors were unwilling to cooperate with other vendors (or other departments within the same vendor’s organisation) to aid problem isolation. It was suggested that *Attributing Accountability* to the source of problems in systems comprising of fewer components and other
integrated parts would be simpler as a consequence of the fewer number of factors to consider.

However, from the interview data it was not just the number of components which affected the ability of Attributing Accountability to problems. The number of involved parties and complexities of customer and vendor relationships also contributed. For example, in systems containing greater numbers of components, supplied by the same vendor, where good working relationships existed between customer and vendor and where vendors willingly cooperated to aid problem resolution the challenge of Attributing Accountability was lessened if the vendor assumed a Degree Of Responsibility.

Denying Responsibility is an ORGANISATIONAL ISSUE as it related to organisations Denying Responsibility for problems.

It was seen that system developers and maintainers could have little influence over other parties Denying Responsibility. Therefore, in such cases, Denying Responsibility by one organisation was Beyond [the] Sphere Of Influence of other organisations.

5.8.2.6 Concept Conflicting Business Motives

Conflicting Business Motives incorporated the conflict between the business motives which occurred between different organisations. For example, there was a conflict between the business motives of COTS component vendors and buyers of COTS components (system developers). COTS component vendors normally aimed to profit by developing and selling components. However, vendors may withdraw components when they did not have enough customers for that product (keeping a component on the market required investment on their part. They may need to fix problems, produce patches, evolve it etc.). However, customers of component vendors, such as system developers, aimed to purchase components to solve business problems, thus, making their profit from the systems they developed (either by developing a system on commission or selling the system etc.). Therefore, a consequence of vendors withdrawing critical components (to satisfy their business motives) would conflict with the business motives of system developers and maintainers.
Conflicting Business Motives can be applied to other business domains, where one organisation aims to purchase a commodity from another. In turn, the buyer’s aim is to use the commodity to further their business aims. The seller endeavours to get the highest price, whereas the buyer’s aspiration may be to acquire the commodity at the lowest price, as well as gaining assurance of continuation of supply.

Conflicting Business Motives related to ORGANISATIONAL ISSUES because ‘conflicting’ business motives normally occurred between different organisations.

The consequence of Conflicting Business Motives, as related to one party, can be Beyond [the] Sphere Of Influence of other organisations. Thus, when a vendor decides to withdraw a critical component from the marketplace system developers and maintainers may not be able to influence this decision.

5.8.2.7 Concept Resisting Change

From the data, this concept emerged as a consequence of people Resisting the Change associated with adopting the COTS-based approach. The culture within some organisations was to accept change with little resistance. However, in other organisations, when staff members were used to specific ways of working, system developers could encounter resistance to change when implementing different systems. Therefore, with reference to Glaser’s (1978, p. 75) theoretical codes it can be seen that Resisting Change was defined in terms of degree, which ranged from Resisting Change to a small degree, through to RESISTING CHANGE by larger degrees.

However, it was suggested that Resisting Change was not only related to the domain of COTS-based development but occurred in any area where people did not willingly embrace change.

Resisting Change was seen to be an ORGANISATIONAL ISSUE because it related to the organisational cultures associated to resisting change.

Resisting Change contributed to Increasing Cost, as a consequence of the additional time and effort required to persuade people to adopt different working practices.
5.8.3 Category DESIGN PRINCIPLES

From Interview A, DESIGN ISSUES appeared to categorise the emerged concepts which related to system design. However, this category was modified to DESIGN PRINCIPLES following further theoretical sorting because this seemed to better categorise the concepts which emerged from the remaining interviews. Thus, from theoretical sorting the following concepts appeared to share the common theme of DESIGN PRINCIPLES:

- Design decision
- Conflicting design decisions
- Vendor homogeneity
- Designing for change
- Design objective
- Balancing design principles
- Conflicting design principles

A diagrammatical representation of the relationship between concepts linking to category DESIGN PRINCIPLES, highlighted in black, is presented in Figure 5.6.

![Diagrammatical representation of the relationship between concepts linking to category DESIGN PRINCIPLES](image)

Figure 5.6: Diagrammatical representation of the relationship between concepts linking to category DESIGN PRINCIPLES.

The relationship between the concepts is presented in the following sections.
5.8.3.1 Concept Design Decision

*Design Decisions* were made by practitioners when designing COTS-based systems. The aim was for *Design Decisions* to support *DESIGN PRINCIPLES*, which are recommendations of how systems should be designed. However, exceptions to this aim occurred when recognised *DESIGN PRINCIPLES* were not possible or deemed unsuitable.

The concept of a *Design Decision* was not specific to COTS-based design. It can be seen that *Design Decisions*, made with reference to *DESIGN PRINCIPLES*, can be applied to any field of design.

5.8.3.2 Concept Designing For Change

COTS-based systems were susceptible to change factors considered *Beyond [the] Sphere Of Influence* of system developers, maintainers and administrators. For example, change factors related to *Change* requirements being ‘forced’ onto system practitioners by the following:

Component vendors: ranged from vendors recommending component upgrades and patches in order for the product to remain in support through to vendors withdrawing components altogether, forcing system practitioners to identify and implement alternative components.

System owners: ranged from system owners implementing different system requirements in order to support new business opportunities.

A consequence of *Change* also affected system stability. For example, changing COTS or other system components could cause a system not to function as before the change activity. Further consequences are that the results of change require testing and problems arising from change need resolving. All of which require human effort to perform. Therefore, it was seen from the data that implementing the concept *Designing For Change* was a key requirement for COTS-based systems designers in order to deliver systems which would suffer minimal impact from *Change* or had a minimal requirement for system change.
It was seen that *Designing For Change* involved designers making *Design Decisions* employing **DESIGN PRINCIPLES** to allow systems to be changed.

### 5.8.3.3 Concept Vendor Homogeneity

*Vendor Homogeneity* implied a commonality of components produced by the same vendor. An assumption held by system developers was that components produced by one vendor would support the same architectural standards and be designed to integrate and work together with minimal effort.

A further assumption was that vendors managed and tested the ongoing maintenance of their components to ensure that they continued to function together.

*Vendor Homogeneity* was considered a *Design Decision* when components were acquired from the same vendor because system developers considered that they would integrate and function together.

However, the data indicated that there were *conditions* associated with *Vendor Homogeneity*. As vendor organisations became larger and released more products the complexity of their product support structure tended to increase. In some organisations, products were developed and supported by different teams. Furthermore, in some organisations the different teams did not necessarily cooperate with each other. Thus, the experience of customers attempting to receive support for multiple components supplied by the same vendor could be poor when the vendor’s support organisation lacked cooperation and synergy.

Therefore, *Vendor Homogeneity* was seen to contribute to *Reducing Costs* because of the assumption that the effort required for integrating and maintaining components supplied by the same vendor would be less than the effort required to integrate and maintain components supplied by different vendors.

*Vendor Homogeneity* was seen as a *Design Decision*, applying the **DESIGN PRINCIPLE** of selecting components which were supplied by the same vendor because they were considered to integrate easily together.
5.8.3.4 Concept Conflicting Design Decisions

Conflicting Design Decisions encompassed the notion of Design Decisions, which when considered individually related to sound design principles, but when implemented together resulted in conflicting consequences. For example, the intended consequence of one design decision may have been to reduce development costs. The assumed consequence of another decision may have been reduced maintenance costs. However, the combination of decisions resulted in increased ongoing costs due to underlying dependencies between components and system parts. Conflicting Design Decisions were occurred as a result of the practicalities and pressures for COTS-based system designers to produce working systems.

Thus, implementing Conflicting Design Decisions could result in a Conflict of Design Principles; the benefit of one decision could be outweighed by other decisions.

Furthermore, there were examples of designers making Conflicting Design Decisions and consciously Balancing Design Principles in order to produce systems which worked and satisfied system and customer requirements.

5.8.3.5 Concept Conflicting Design Principles

There are different COTS-based system design principles. However, the basis of concept Conflicting Design Principles was that the benefit of one design principle outweighed or resulted in a conflict when combined with other design principles.

Thus, Conflicting Design Decisions inadvertently resulted in Conflicting Design Principles. However, ‘conscious’ Design Decisions were made whereby Conflicting Design Principles resulted in Balancing Design Principles. The practicalities of the real-world required system designers to make conscious decisions, resulting in Balancing Design Principles, in order to build functioning COTS-based systems with reference to the system requirements and available components.
5.8.3.6 Concept Design Objectives

The Design Objectives of commercial COTS-based system designers were to produce systems which solved business problems and addressed requirements, with reference to the availability of components. Design Objectives were satisfied by designers making Design Decisions, normally with reference to DESIGN PRINCIPLES. However, a consequence of the practicalities of the real world, such as the availability of suitable components, vendor cooperation and customer requirements, resulted in Design Objectives being addressed by Design Decisions leading to Conflicting Design Principles or by designers Balancing Design Principles.

5.8.3.7 Concept Balancing Design Principles

There were different DESIGN PRINCIPLES associated with COTS-based system design. These varied from selecting components which were highly cohesive, but loosely coupled in order to allow components to be changed in isolation of other system parts; designing systems using components supporting the same architectural standards and exploiting the flexibility of the COTS-based design approach by sourcing components from different vendors.

However, related to the feasibility of building COTS-based systems in the real world designers were forced into Balancing Design Principles in order to produce functioning systems. Thus, the Design Decisions underpinning Balancing Design Principles tended to arise from conscious decisions made by system designers. This differed from Conflicting Design Principles, which occurred as a consequence of Design Decisions. Thus, with reference to Glaser’s (1978, p. 76) strategy theoretical coding family Balancing Design Principles were a strategy for handling the design challenges of building COTS-based systems. The deficiencies of one design principle were balanced by the benefits of other design principles.

However, Balancing Design Principles could still result in Conflicting Design Principles because in attempting to Balance Design Principles designers were left with design principles which conflicted. An example of this was where customers forced system designers to select inappropriate components. System designers attempted to balance the effect by applying other design principles. However, the net effect was a conflict of design principles.
5.8.4 Category MANAGING COMPLEXITY

From Interview A category COMPLEXITY ISSUES was deemed to categorise concepts relating to System Complexity. However, this category was modified MANAGING COMPLEXITY because as further concepts emerged from other interviews it was apparent that complexity was not only related to system complexity. Complexity also encompassed the complexity of relationships between people. Thus, it appeared that factors and processes relating to MANAGING COMPLEXITY better described the relationship between the ‘complexity’ related concepts because assessing, controlling and managing complexity appeared to be contributing factors to CONTROLLING COST.

Thus, from theoretical sorting the following concepts appeared to share the common theme of MANAGING COMPLEXITY:

- Maintenance complexity
- System complexity
- Degree of dependency
- Relationship complexity
- Attributing accountability
- Denying responsibility

A diagrammatical representation of the relationship between concepts linking to category MANAGING COMPLEXITY, highlighted in black, is presented in Figure 5.7.
The relationship between the concepts represented in Figure 5.7 is presented in the following sections.

5.8.4.1 Concept System Complexity

*System Complexity* conceptualised the number of interrelated factors making up a systems. *System Complexity* can be applied to any system where interrelating factors occur. However, for the purposes of this study *System Complexity* relates to COTS-based systems.

With reference to Glaser’s (1978, p. 75) *degree* theoretical coding family, *System Complexity* can be viewed in terms of a continuum, ranging from a low to high degrees of *System Complexity*. Furthermore, there were factors and processes which contributed to *Reducing or Increasing System Complexity*.

From the interview data it was seen that *System Complexity* was related to different factors:
The number of different connections. Thus, systems containing greater numbers of components and interconnected interfaces contributed to ‘Increasing System Complexity.

The number of dependencies. Systems comprised of more dependencies were defined in terms of higher degrees of System Complexity because changing one system part required further changes to other system parts. Therefore, the Knock-On Effect of seemingly small changes was greater levels of change activity.

Amount of functionality. Components providing greater amounts of functionality contributed to Increasing Complexity, compared with components providing simple instances of functionality. Thus, combining complex components contributed to increasing System Complexity.

As with cost, examples from the data indicated factors relating to either increasing or reducing System Complexity. However, there was no indication of a baseline level of System Complexity to which increasing or reducing was compared against.

System Complexity relates to category MANAGING COMPLEXITY because system designers, developers and maintainers endeavoured to manage system complexity when designing, developing or maintaining COTS-based systems. Complexity was a contributing factor of Cost. A consequence of Increasing Complexity was Increasing Cost because of the number of factors to be considered, requiring more time and higher levels of skill. Thus, MANAGING COMPLEXITY involved decisions and actions, with the aim of Reducing Complexity as a means of Reducing Costs.

5.8.4.2 Concept Maintenance Complexity

Maintenance Complexity conceptualised the number of interrelated tasks, relationship and dependencies between tasks, which were performed when maintaining COTS-based systems. For the purpose of this study, maintenance encompassed tasks occurring after system implementation.

With reference to Glaser’s (1978, p. 75) theoretical coding families Maintenance Complexity could be measured in terms of degree. For example, maintenance tasks only
involving changing single components with no underlying dependencies can be seen as possessing lower degrees of *Maintenance Complexity*. Conversely, maintenance activity involving many components, with higher degrees of dependencies, requiring people with specialist skills can be seen to be associated with higher degrees of Maintenance Complexity.

Furthermore, with reference to Glaser’s (1978, p. 74) *Six C’s* theoretical coding family the expectation was that *Maintenance Complexity* was influenced by *System Complexity*, rather than the other way round. Thus, higher degrees of *System Complexity* contributed to increasing degrees of *Maintenance Complexity*.

5.8.4.3 Concept Degree Of Dependency

Concept *Degree Of Dependency* was defined in terms of the number of interrelated dependencies between COTS components and system parts and the effect on system functionality and stability resulting from making changes to one or more system parts.

With reference to Glaser’s (1978, p. 75) *degree* theoretical coding family *Degree Of Dependency* could be considered in terms of a continuum, ranging from *low* through to *high Degrees Of Dependency*. The higher the *Degree Of Dependency* between greater numbers of system parts the greater the contribution to *Maintenance Complexity* and *System Complexity* as a result of the number of dependent factors to be considered. As *Degree Of Dependency* increased the ability to change one system part, in isolation of changing other system parts, diminished.

*Degree Of Dependency* is a factor of **MANAGING COMPLEXITY** because *Dependency* contributes to *Complexity*. Thus, the higher the *Degree Of Dependency* between components, the more factors to be taken into account by practitioners tasked with managing complexity.

5.8.4.4 Concept Relationship Complexity

*Relationship Complexity* conceptualised the complexities of human relationships, as opposed to technical complexities, which were encompassed by *system* or *maintenance* complexity. Contributing factors to *Relationship Complexity* concerned the number of vendors and support organisations system developers and maintainers had to deal with.
Therefore, systems comprising of components supplied from different vendors were associated with higher degrees of *Support Complexity* for system maintainers, as a result of the number of different support organisations and support personnel which had to be dealt with in order to receive support for the system as a whole.

The ability to deal with *Relationship Complexity*, throughout a system’s lifespan, was a factor of **MANAGING COMPLEXITY** as a result of system developers and maintainers having to deal with and negotiate with different populations, including vendors, support teams and customers.

With reference to Glaser’s (1978, p. 75) theoretical codes *Relationship Complexity* could be defined in terms of *degree*, which ranged from *lower* to *higher* degrees of *Relationship complexity*. Higher degrees of *Relationship Complexity* had a greater influence on **MANAGING COMPLEXITY** as a result of the number of populations to be dealt with.

There was an inverse relationship between *Relationship Complexity* and *Support Quality* because as the complexity of support relationships increased the quality of support received decreased as a result of greater numbers of people to be dealt with.

Furthermore *Denying Responsibility* compounded *Relationship Complexity*. For example, in situations where system developers or maintainers were required to deal with different support organisations for problem resolution the consequence of vendors *Denying Responsibility* for problem root cause added to *Relationship Complexity* and contributed further to the effort required in **MANAGING COMPLEXITY**.

**5.8.5 Category MANAGING CHANGE**

Category **MANAGING CHANGE** emerged from the theoretical sorting of concepts, which originated from all of the interviews. From the data the concept of *Change* was suggested to have a greater effect on COTS-based systems, than with custom-built systems, as a consequence of component vendors imposing the requirements to change their components. This occurred in addition to *Change* being driven by variations in business focus and customer requirements, which also affected custom-built systems.
However, closer examination of the concepts relating to *Change* indicated that COTS-based system developers or maintainers had to deal with the effects of *Change*. Thus, **MANAGING CHANGE** seemed to better categorise the change-related concepts. This was because greater proportions of the effort and cost associated with maintaining COTS-based systems occurred as a result of planning for and implementing *Change*.

Some of the concepts which emerged from the data related to *Design Principles* which facilitated *Change*. The aim was for *Design Principles* to be applied at the beginning of COTS-based projects because was considered unfeasible to retro-fit these principles once systems were implemented. The remaining concepts can be seen to explain the consequences of change on system stability.

Thus, from theoretical sorting the following concepts appeared to share the common theme of **MANAGING CHANGE**:

- Managing change
- Designing for change
- Implications of change
- Change unpredictability
- Knock-on-effect
- Resisting change

A diagrammatical representation of the relationship between concepts linking to category **MANAGING CHANGE**, highlighted in black, is presented in Figure 5.8.
The relationship between the concepts in Figure 5.8 is presented in the following sections.

5.8.5.1 Concept Managing Change

From the data a certainty was of Legacy implementation[s] PMiH1 and other entities Changing over time PMiH2. Most software systems change over time, Operating systems require upgrading, COTS-components evolve over time or are withdrawn, business requirements change and new business opportunities arise. Therefore, Managing Change was required by system developers and maintainers to react to the requirement of Changing over time PMiH2 in order to keep systems functioning.

Managing change was a contributing factor of CONTROLLING COST because it involved effort, and in many cases, specialist skills. Thus, in order to successfully ‘control costs’ Managing Change was required to be performed effectively.

Changing COTS-based systems, and the costs of Managing Change, were contributing factors to Increasing Costs as a result of the effects of dealing with unknown factors. These factors were deemed to have been unpredictable when the systems were originally conceived. An example of this was when a vendor released a new version of a component, which in turn required an operating system upgrade.
5.8.5.2 Concept Designing For Change

The purpose of *Designing For Change* was to make **MANAGING CHANGE** easier for system developers and maintainers. *Designing For Change* involved applying *Design Principles*, which allowed parts of COTS-based systems to be changed with minimal impact to rest of the system.

Thus, with reference to Glaser’s (1978, p. 74) Six C’s theoretical coding family the perceived *consequence* of implementing a *Designing For Change* policy was to facilitate the process of **MANAGING CHANGE** (i.e. contribute to reducing cost, being less onerous). This enabled system maintainers to change system parts with little effect on the rest of the system. However, *Designing For Change* was normally required to be considered early on in the design process because it tended not to be possible to retro-fit *Design Principles* after systems were implemented.

Conversely, when changing system parts the consequence of the absence of a *Designing For Change* policy could lead to a *Knock-On Effect* - the requirement to change other system parts.

5.8.5.3 Concept Implications Of Change

The process of **MANAGING CHANGE** required an understanding of the *Implications Of Change* because *Change* resulted in different implications, such as, one change requiring additional change activity as a result of dependencies.

5.8.5.4 Concept Knock-On-Effect

Dealing with the *Knock-On-Effect*[s] of *Change* contributed to the process of **MANAGING CHANGE**. **MANAGING CHANGE** involved dealing with the *Knock-On-Effect*[s] of change because performing one change action often required additional change activity, which all required managing.

The *Knock-On-Effect* of *Change* was also viewed as an *Implication Of Change*. For example, when the *Knock-On-Effect* of one change was further change activity this was viewed as an *Implication* of the initial change.
5.8.5.5 Concept Change Unpredictability

Change Unpredictability made MANAGING CHANGE more challenging for system maintainers, because when change activity could not be predicted the ability to control the cost of MANAGING CHANGE was diminished. For example, the norm for commercial system designers and developers was to estimate the total lifecycle costs of systems before developing the systems. Profit margins can be calculated from these estimates. However, Change Unpredictability rendered these estimates inaccurate because additional ‘unpredicted’ change activity added extra cost, not accounted for in the initial estimates.

Furthermore, because of the Unpredictable effect of Change on system parts Change Unpredictability was also seen to be an Implication Of Change.

5.8.5.6 Concept Resisting Change

Resisting Change differed from the other concepts linked to category MANAGING CHANGE because it related to the human reluctance to accept change. The other concepts emerged from technical issues relating to change.

Resisting Change related more to the aftermath of change, rather than to the process of change itself (there may be instances of resistance to the change process, however, this did not emerge from the data collected for this study). For example, there was resistance to business processes changing, occurring as a result of implementing COTS-based solutions in place of the original bespoke systems.

Therefore, the processes related to MANAGING CHANGE should be extended to deal with Resisting Change in order for any cost benefits related to implementing change to be realised. Thus, when people chose not to adopt and utilise changed items then any planned benefits related to the implementation of these changes would not occur.

5.8.6 Category CULTURAL ISSUES

Cultural issues emerged from the interview data as a result of the varying quality in which vendors published details of the functionality of their components. However, due
to a variety of reasons, such as their country of origin and other cultural factors, there was an inconsistently in the way different vendors described the capabilities of their components. For example, the functional specifications of components tended to be described in the English language. When English was not the primary language of technical writers this resulted in inconsistencies in the description of component functionality. Furthermore, there were differences in the way some English words were interpreted between people originating from English speaking countries, such as the United Kingdom and the United States of America.

**CULTURAL ISSUES** made it difficult for system developers to assess component suitability as a result of some vendors describing component functionality differently from other vendors. Thus, the effect on **Cost** was that system developers had to spend more time establishing component suitability, in order to select appropriate components. Time was related to cost.

*Culture* was taken to mean: “the collective programming of the mind which distinguished the members of one group or category of people from those of another and applied to nations, organisations, occupations, and professions” (Hofstede, 1994, p. 4). From the interview data the assumption was that cultural relativity, the culture of the human environment in which an organisation operated (Hofstede, 1994), influenced the level of support customers (in this case, system developers are the customers of component vendors) received from vendors. In different cultures the definition of what constituted good customer service varied.

For the purpose of this study **CULTURAL ISSUES** related to aspects associated with national and belief cultures. This differed from **ORGANISATIONAL ISSUES** which related to issues affecting organisations, organisational structure and the relationship between organisations. However, it can be seen that some organisations may differ from others as a result of **CULTURAL ISSUES**.

From theoretical sorting the following concepts appeared to share the common theme of **CULTURAL ISSUES**:

- Cultural misunderstanding
- Appreciating cultural factors
• Lacking common understanding

A diagrammatical representation of the relationship between concepts linking to category CULTURAL ISSUES, highlighted in black, is presented in Figure 5.9.

![Diagrammatical representation of the relationship between concepts linking to category CULTURAL ISSUES.](image)

Figure 5.9: Diagrammatical representation of the relationship between concepts linking to category CULTURAL ISSUES.

The relationship between the concepts represented in Figure 5.9 is presented in the following sections.

**5.8.6.1 Concept Cultural Misunderstandings**

*Cultural Misunderstandings* conceptualised that the consequence of cultural differences could result in misunderstandings. One example of this was the difference in the use and interpretation of language between cultures.

Within the data, the link between open codes *COTS supplier issues PMiA9, Multiple vendors PMiA10* and *Difficulty understanding terminology PMiA13* were *Cultural Misunderstandings* as a result of system developer’s *Difficulty understanding terminology PMiA13*. This was influenced by the cultural differences of multiple vendors, who potentially, originated from anywhere in the world.

With reference to Glaser’s (1978, p. 75) theoretical coding families the effect of *Multiple vendors PMiA10 on Cultural Misunderstandings* could be measured in terms
of degree. The consequence of using greater the numbers of components, sourced from multiple vendors, based in different countries, resulted in a greater chance of system designers, developers and maintainers misinterpreting the published details of component functionality as a result of Cultural Misunderstandings.

The impact of Cultural Misunderstandings could be alleviated by Appreciating Cultural Factors because the result of being conscious that cultural factors existed was considered to influence the understanding of others, which helped to address Misunderstandings.

Cultural Misunderstandings share the common theme of CULTURAL ISSUES because the basis of the concept relates to cultural influences.

5.8.6.2 Concept Appreciating Cultural Factors

Appreciating Cultural factors conceptualised that cultural differences existed between people originating from different countries, which included their use of and interpretation of language. Appreciating Cultural Factors assisted people in dealing with the affect of cultural differences. For example, by consciously appreciating of the existence of cultural factors, system designers, developers and maintainers may be less likely to accept technical descriptions of components at face value and support the investigation of component functionality with other methods.

The basis of concept Appreciating Cultural Factors can be seen to share a common categorisation of CULTURAL ISSUES because it relates to human cultural factors.

5.8.6.3 Concept Lacking Common Understanding

Lacking Common Understanding conceptualised that factors relating to building COTS-based systems may not be understood in the same way by all parties. This occurred as a result of the cultural differences which existed between people. Lacking Common Understanding occurred as a result of Cultural Misunderstandings because of the way people from different countries and cultures may interpret factors.
Lacking Common Understanding can result from CULTURAL ISSUES, such as, different interpretations of language and behaviour occurring as a consequence of the differences between people from different nationalities.

5.9 Relationships between Categories

The aim of this section is to examine the relationship between the categories which emerged from the analysis of the interview data. Glaser (1978; 1992; 1998; 2001) stated that the articulation of the relationships between categories forms the basis of a Grounded Theory.

From the analysis the following categories emerged:

- CONTROLLING COST
- ORGANISATIONAL ISSUES
- DESIGN PRINCIPLES
- MANAGING COMPLEXITY
- MANAGING CHANGE
- CULTURAL ISSUES

Figure 5.10 provides a visual representation of the relationship between categories.
The relationship between the categories, as represented in Figure 5.10, is discussed below.

The main focus of the interview data was Cost. System designers, developers and maintainers, when asked for their views on building system from COTS components, did so with reference to Cost. The consensus was that there was cost associated with building software systems. This is not new. However, one reason for choosing to build systems from COTS software components was that this method was perceived to contribute to reducing costs, compared with building systems from scratch. Furthermore, system developers understood that there were different cost challenges associated with building systems from COTS components as well as ‘good practice’ design principles.

However, for each COTS-based system it may not be possible to implement all of the recommended design principles in order to minimise development and maintenance costs. From the data a common pattern was that system designers, developers and maintainers endeavoured to control costs, in relation to the available options and cost challenges. For example, a chosen Design Principle may be perceived to contribute to reducing developments costs. However, a consequence of other factors related to the Design Principle may be increasing maintenance costs. Therefore, system developers
and maintainers were attempting to control the effect of different factors which influenced the cost of developing and maintaining COTS-based systems.

The properties of core category CONTROLLING COST supported this. These were concepts relating to Increasing Cost, Cost Reducing Strategies and processes aimed at Balancing Cost Challenges of the COTS-based system approach. Thus, by system designers, developers and maintainers understanding that there were issues contributing to Increasing Costs, which could be addressed by decisions aimed at Balancing Cost Challenges and the implementation of Cost Reducing Strategies the end result contributed to CONTROLLING COST.

CONTROLLING COST emerged as the core category as a result of Cost emerging as the main focus from the interview data and identified as an area which must be controlled.

The properties of category MANAGING COMPLEXITY related to technical complexities of system and maintenance and human relationship complexity. A consequence of ‘increasing’ Complexity was higher costs as a result of the number of factors which needed to be considered, which required additional effort, people, skills and time. Thus, one focus of MANAGING COMPLEXITY related to system designers, developers, maintainers attempting to ‘reduce’ Complexity. Thus, a consequence of MANAGING COMPLEXITY ‘effectively’ was ‘reducing’ Cost, and thus, contributed to CONTROLLING COST. Conversely, failing to ‘manage’ Complexity effectively was detrimental to CONTROLLING COST because development and maintenance costs were likely to rise.

Other factors related to CONTROLLING COST also influenced the options for MANAGING COMPLEXITY. For example when sufficient resources (monetary budget) were provided a wider choice of Design solutions were considered more likely to be available for selection, thereby contributing to ‘minimising’ Complexity.

Change was a contributing factor to Increasing Cost. It was seen from the interview data that systems which were subjected to ‘less’ Change incurred ‘less’ Cost, compared with systems which were subjected to greater amounts of change activity. In addition to the additional effort and planning required to action change activity, change could
compromise system stability, which necessitated additional remedial work. With reference to Glaser’s (1978, p. 75) theoretical codes the relationship between Cost and Change can be explained in terms of degree. A consequence of higher degrees of Change tended to be higher costs and thus, lower degrees of change, tended to result in lower costs. Thus, MANAGING CHANGE effectively, whereby changing system parts had minimal impact on other system parts, contributed to CONTROLLING COST.

However, MANAGING CHANGE for COTS-based systems, compared with custom built systems, differed because Change could be ‘forced’ upon system maintainers by vendors supplying upgraded components. However, it can be seen that other instances of Change can be forced on designers or maintainers of custom-built systems, such as the requirement to add new functionality. However, this type of change activity tended to be requested by customers who would bear the cost.

Category DESIGN PRINCIPLES encompassed concepts relating to COTS-based system design. System designers made decisions aimed at applying DESIGN PRINCIPLES when designing COTS-based systems. However, a consequence of making Design Decisions in the real world could result in Conflicting Design Principles because the ‘ideal’ design options may not be available or feasible for each system.

However, the application of DESIGN PRINCIPLES was considered to assist COTS-based system developers with CONTROLLING COSTS by minimising Complexity and facilitating Change. For example, the design principles of building systems from as few components as possible, which were supplied by the same vendor and which supported the same architectural standards aimed to contribute to ‘reducing’ System Complexity by reducing the number of separate factors to be considered and managed. Furthermore, selecting components which were loosely coupled, with lower degrees of underlying dependencies, reduced Maintenance Complexity by enabling the ability for components to be changed in isolation of changing other system parts.

The assumption of system developers was that the implementation of DESIGN PRINCIPLES enhanced their capability to MANAGE COMPLEXITY because one aim of good system design was to ‘minimise’ Complexity. However, with reference to Glaser’s (1978, p. 75) degree theoretical coding family the relationship between
applying **DESIGN PRINCIPLES** and their effect on **MANAGING COMPLEXITY** could be defined in terms of *amount*. Making *Design Decisions* where greater proportions of **DESIGN PRINCIPLES** were implemented reduced system and maintenance complexity. ‘Good’ *Design Decisions* also involved sourcing components from a minimum number of vendors, in addition to building good relationships with vendors’ support teams, in order to reduce *Relationship Complexity*. However, a consequence of including lower amounts of ‘sound’ **DESIGN PRINCIPLES** within *Design Decisions* could render **MANAGING COMPLEXITY** more challenging.

With technical solutions, **MANAGING COMPLEXITY**, by applying **DESIGN PRINCIPLES** and avoiding Conflicting Design Principles, enabled *Costs* to be more efficiently *Controlled* as a result of leaving fewer cost factors to chance. However, **ORGANISATIONAL ISSUES** and **CULTURAL ISSUES** introduced variables with which less control could be attributed. Thus, the effect of **ORGANISATIONAL ISSUES** and **CULTURAL ISSUES** on **MANAGING CHANGE** and **CONTROLLING COST** were less predictable. For example, **ORGANISATIONAL ISSUES**, where vendors were unable or unwilling to provide assistance with the use of their components, could be *Beyond [the] Sphere Of Influence* of system designers, developers and maintainers. In such cases system designers, developers and maintainers had little influence on how vendor organisations conducted their business. Thus, issues which were *Beyond [the] Sphere Of Influence* could contribute to *Increasing Cost* as a result of the additional time and effort required to resolve problems. Thus, **ORGANISATIONAL ISSUES** could influence *Cost* and the ability of practitioners to ‘control’ *Cost*.

Furthermore, with *Support Quality* system developers and maintainers were normally unable to *force* support organisations, from different support organisations, to change the levels of *Support Quality* provided. Additionally, system developers and maintainers were not normally in a position to change component suppliers, in response to poor support quality, due to contractual obligations or the availability of alternative components delivering the required functionality.

**CULTURAL ISSUES** requires further investigation because the data indicated that this was an area which influenced *Cost* by introducing unpredictability as a result of the way people from different cultures interpreted factors. With the global distribution of
software, **CULTURAL ISSUES** may increasingly shape the ability of system designers to successfully control cost. Producers of COTS components and technical documentation may originate from different cultures, compared with those who purchase the components. Thus, the consequence of buyers of COTS components misinterpreting documented component functionality, can contribute to *Increasing Costs* as a result of the additional effort required to modify components or to source alternative products.

### 5.10 Grounded Theory Output – Principles Relating to the Issues Affecting COTS-Based System Development

Glaser (1978; 1998; 2001) advised that a ‘Grounded Theory’, which explained and predicted real world actions, would emerge as a result of following his Grounded Theory method; the Grounded Theory would be articulated from the relationships between concepts and categories. However, for this part of the study, although there were links between the categories, the relationships between categories did not indicate an obvious *theory*.

**CONTROLLING COST** emerged from the data as the core category because it conceptualised the underlying processes performed by system developers and practitioners when endeavouring to control the costs of developing and managing COTS-based systems. In the absence of a clear *Grounded Theory* it was felt that the relationships between **CONTROLLING COST** and other emergent concepts and categories could be articulated as a set of principles, rather than as a formal theory. These are presented below and referred to as *grounded principles* because they were developed from the Grounded Theory analysis.

### 5.10.1 Grounded Principles

The principles and justifications are as follows:

a) **Complexity can affect the total lifecycle costs of COTS-based systems and the ability of practitioners to control costs.** This was supported in Figure 5.10: *A visual representation of the relationship between categories*, where it can be seen that category **MANAGING COMPLEXITY** was a factor of **CONTROLLING COST**. Thus, to influence system costs factors affecting
complexity should be addressed. The use of COTS components to build systems should reduce ‘technical’ Complexity as a result of vendors developing the functionality provided with components. However, from the perspective of system developers using COTS components, Complexity comprised of factors relating to System Complexity and Relationship Complexity. System Complexity was defined by the number of different components making up a system, the number of different vendors involved in supplying system components, the degree of dependency between components, the underlying architectural standard of the components and the amount of effort required to tailor and integrate components. Maintenance Complexity was influenced by System Complexity. Therefore, Maintenance Complexity was likely to increase as System Complexity increased. Relationship Complexity was concerned with the human factors affecting complexity. As the number of vendors supplying components to a system increased so did the complexity of the relationships which had to be developed with the different support organisations. The consequence of involving greater numbers of parties or organisations was an increase in the opportunities for people to question Accountability and to Deny Responsibility for problems and issues with their products. There was also a likelihood that the overall support quality would diminish as the number of support groups increased.

b) **The application of Design principles are required to reduce complexity, enable system change and reduce total lifecycle costs of COTS-based systems.** However, the problem was in identifying the appropriate COTS-based system design principles. For example, building systems with fewer COTS components was proposed as a Design Principle because it was considered to reduce the number of connections between components requiring management. However, in isolation, this principle may not ‘reduce’ Complexity or Cost when the components do not support the same architectural standard or were not supplied by the same vendor. Thus, greater numbers of components, supporting the same architectural standard and which were supplied by the same vendor, had a greater effect on Reducing Complexity and Cost because the components tended to integrate easier, required less effort to upgrade and involved fewer support organisations when dealing with problems. Therefore, the issue was concerned with defining the best balance of Design Principles. However, from
the interview data it was found that Design Principles were required to facilitate the production of COTS-based systems, which supported the requirements, required minimal integration effort and allowed system parts to be changed in isolation of the other system parts. This view is displayed in Figure 5.10: A visual representation of the relationship between categories, where it can be seen that category DESIGN PRINCIPLES is a factor of categories MANAGING CHANGE and MANAGING COMPLEXITY and of the core category CONTROLLING COST.

c) The maintenance costs of COTS-based systems may be higher than custom-built systems. With reference to Figure 5.4: Diagrammatical representation of the relationship between concepts linking to category CONTROLLING COST, there are numerous cost factors which were considered to contribute to the costs to maintain COTS-based systems. Furthermore, with reference to the interviews conducted for the Grounded Theory analysis (see 5.8.1 Category CONTROLLING COST) Change, Change Unpredictability and the Implications Of Change contributed to maintenance costs rising as a result of the effort required to implement change and to deal with any ‘unpredictable’ Implications Of Change. In some instances, changing COTS-based systems required Redoing Integration effort. Furthermore, with reference to Sections 5.8.4 Category MANAGING COMPLEXITY and 5.8.5 Category MANAGING CHANGE it was suggested that over time, the combination of the requirements of MANAGING COMPLEXITY and MANAGING CHANGE contributed to the costs to maintain COTS-based systems. As Complexity increased and more Change activity was required, additional effort was incurred assessing the impacts of change and performing change actions. Change tended to occur during the maintenance phases of the system lifecycle. It was suggested from the interview data that the combination of ‘inappropriate’ design decisions, high degrees of complexity, frequent change activity, strained relationships with vendors, poor product support and high degrees of dependencies between components would result in the maintenance costs being higher than for custom-built systems.

d) Designing systems in a way which allows components to be changed independently of other system parts is required to reduce ongoing costs of
COTS-based systems. One publicised benefit of CBD was that this method should reduce system development costs (Bhuta et al., 2005; Li et al., 2006; Lin et al., 2007). However, over time most systems tended to change. For example, changes of components were often forced on customers by vendors and components were withdrawn by vendors. Additionally, systems requirements changed, requiring functionality to be added or modified. Over a period of time, such as ten years, it was suggested in the interview data that the resultant system would be totally different for the system originally developed. Thus, in order to reduce ongoing costs COTS-based systems should comprise of loosely coupled components with minimum degrees of dependency with other system parts. The objective of this principle was to minimise the effort required to change system parts and to minimise the consequence of change on system stability by enabling the changing of system parts in isolation of changing other parts. This view is supported in Figure 5.7: Diagrammatical representation of the relationship between concepts linking to category MANAGING COMPLEXITY, where it can be seen that the Degree Of Dependency contributed to Maintenance Complexity. Furthermore, with reference to Figure 5.10: A visual representation of the relationship between categories, it can be seen that category MANAGING COMPLEXITY was a contributing factor to CONTROLLING COSTS.

e) To reduce total lifecycle costs the functionality of COTS-based systems should be supplied from components which support the same architectural standard. This principle was conceptualised in category DESIGN PRINCIPLES. From the interview data the number of components making up a system was not considered to be the main cost factor. In some cases, the selection of a greater number of components, supporting the same architectural standard, contributed further to reducing the ongoing costs of COTS-based systems, compared with using fewer components which supported different architectural standards. This was justified from the assumption that components which supported the same architectural standard were likely to integrate with less effort and require less ongoing effort to maintain. However, where possible, systems should be constructed from the minimum number of components (which also support the same architectural standard). It should also be noted that from the data it was considered that components supplied by the same vendor
tended to be built to integrate together. Thus, it was deemed preferable to source as many components from the same vendor. However, it was also appreciated by the interviewees that a vendor could produce a range of components which supported different architectural standards.

f) **To reduce total lifecycle costs the functionality of COTS-based systems should be supplied from components which are supplied by the minimum number of vendors.** The basis of this principle was related to categories MANAGING COMPLEXITY and ORGANISATIONAL ISSUES. From the interview data, reducing the number of vendors supplying components to a system contributed to ‘reducing’ Relationship Complexity because fewer support organisations needed to be contacted when diagnosing system problems. Furthermore, minimising the number of vendors supplying components was considered to reduce the opportunities for vendors to ‘deny’ Responsibility for problems occurring within a system.

g) **The costs to develop and maintain COTS-based systems are not only associated with technical issues. Human problems relating to relationships and cultural influences can affect the total lifecycle costs of COTS-based systems.** With reference to Figure 5.10: A visual representation of the relationship between categories, it was seen that both ORGANISATIONAL ISSUES and CULTURAL ISSUES affected Cost and the ability to Control Cost. Thus, human factors arising from organisational and cultural issues added to the complexities of CONTROLLING COST. However, additional research needs to be conducted into the affect of Cultural Issues on CBD.

h) **It may not be possible accurately predict the total life costs of COTS-based systems as a consequence of the uncertainties in the COTS marketplace and probable changes in system requirements.** The probability that vendors will continue to produce and support the same components or that system requirements will remain the same over time was considered low. The consequences of these factors made it difficult for system developers to accurately estimate the total life costs of COTS-based systems. In scenarios where system developers have to submit bids in order to win system development or system management opportunities, the ability to accurately
estimate lifecycle costs was considered crucial. The submission of bids which are too low may result in winning a system development or management opportunity, but challenge the profitability of a project. From the interview data, *Change* was considered to be a factor which will affect all systems; over a ten year period all COTS-based systems were deemed likely to significantly change. However, although it can be seen that *Change* affects all system types, not just COTS-based systems, the differentiating factor between the system types related to the fact that the COTS components were supplied by a *vendor*, who tended not to be part of the same organisation. Thus, over time, vendors will invariably change components or withdraw them from the marketplace, necessitating the modification of systems. With reference to Figure 5.8: *Diagrammatical representation of the relationship between concepts linking to category MANAGING CHANGE*, concept *Designing For Change* was a factor of MANAGING CHANGE. Support for this principle was also partially provided from the results of the statistical analysis (see Chapter 4) which suggested that the measurement of Function Points only constituted a weak predictor of the effort required to develop and maintain systems containing COTS components. However, figures representing maintenance effort were only available for a five year period.

i) **Developing working relationships with vendor support organisations and internal support teams can contribute to reducing costs.** The concept of *Support quality* was considered to be a *Cost* contributing factor. Thus, when problems occurred with COTS-based system development or maintenance the cooperation of support teams was deemed crucial in order to resolve issues in a timely manner, as the time taken to fix problems affected *Cost*. Therefore, good working relationships should be developed with all support organisations and teams in order to facilitate the timely resolution of issues. However, with reference to Figure 5.5: *Diagrammatical representation of the relationship between concepts linking to category ORGANISATIONAL ISSUES*, receiving the appropriate *Support Quality* could be *Beyond The Sphere Of Influence* of system developers. Some vendors were not willing to offer appropriate product support. With larger vendor organisations, different products tended to be supported by different support teams from within the same organisation, which often appeared as different support organisations to customers. Furthermore, internal support
teams often lacked the ability to provide quality support for components as they required input from vendor organisations.

5.11 Conclusions

This chapter has presented the results of the analysis of interview data using the Grounded Theory method. The aim of the Grounded Theory analysis was to investigate the cost factors of CBD using data collected from within IBM. However, the interviewees were unable to provide insider knowledge of the projects analysed as part of the statistical analysis section. This was an unavoidable limitation of the analysis.

The main focus which emerged from the Grounded Theory analysis was the concept of Cost, which evolved into core category CONTROLLING COST. This encompassed the issues, articulated in concepts and categories, which system designers, developers and maintainers considered as important when managing the costs of developing and maintaining COTS-based systems.

Cost emerged as a focus because the choice to select COTS-based solutions in the first place was often made with reference to perceived cost savings resulting from using this method.

Contributing factors to CONTROLLING COST were DESIGN PRINCIPLES, MANAGING CHANGE and MANAGING COMPLEXITY. Thus, when inappropriate DESIGN PRINCIPLES were applied, if MANAGING CHANGE lacked control or if MANAGING COMPLEXITY was ineffective, the ability of system practitioners to ‘control’ Cost was challenged. Furthermore, ORGANISATIONAL ISSUES and CULTURAL ISSUES were also factors of CONTROLLING COST.

Glaser (1978; 1998) suggested that a Grounded Theory would emerge as a result of following his Grounded Theory method. However, an obvious theory did not emerge from this analysis. The reasons for this are discussed in Chapter 7.

However, it was felt that the results of the Grounded Theory analysis could be expressed as a set of principles. These are displayed in Section 5.10.1. The assessment of their applicability is an area for further research.
The next chapter provides a review of the literature to determine if there is support for the results of this Grounded Theory analysis from other sources of research.
Chapter 6 Literature Review to Assess the Applicability of the Results of this Grounded Theory Analysis

6.1 Introduction

Glaser (1978) recommended that a literature review should not be performed until after performing the Grounded Theory analysis. The justification of his recommendation was to avoid the researcher’s views being tainted by preconceptions arising from the literature. However, for this study a literature had been performed before considering the use of the Grounded Theory method.

In order to adhere as much as possible to Glaser’s Grounded Method it was decided to conduct a second literature review. The aim of this section was to determine how the results from the Grounded Theory analysis corresponded with the literature.

The literature review was conducted by searching the following online databases: Web of Knowledge, INSPEC, Web of Science, Compendex, IEEE Electronic Library and ACM Digital library. These data sources were selected because they were deemed to be relevant to this area of research and they were provided by the University of Portsmouth.

The following search terms were used:

“COTS and Cost”; “COTS and Design”; “COTS and Culture”; “Effort”; “Total Cost of Ownership”; “Maintenance cost”; “COTS”; “Functional density rule of thumb”

6.2 Literature Review

The literature review is presented below. It is formatted to show how the literature supported the emergent categories and identified principles.

6.2.1 Controlling Cost

A cost challenge which can impact system designers occurred during the planning phase of COTS-based system project. This related to the requirement for system designers to submit development and maintenance cost estimations early on in a project in order to
win the system development business in the first place (Lamine, Jalani & Ghezala, 2005; Perez et al., 2008).

Thus, the challenge associated with **CONTROLLING COSTS** for system designers and developers related to their ability to estimate integration and ongoing costs of COTS-based system early on in the design process, often before specific components had been selected. The consequence of generating inaccurate cost estimations at the beginning of a project impacted the ongoing profitability of the system (Naunchan & Sutivong, 2007).

By generating an accurate estimation for development and ongoing maintenance costs early on in a project, especially before all component selections had been made, was difficult because the costs, tasks and effort required to integrate and maintain sets of components could vary as a result of problems occurring during the life of a system (Lamine et al., 2005; Perez et al., 2008).

In the Grounded Theory analysis performed in this study concept **Balancing Cost Challenges** emerged as a factor of core category **CONTROLLING COST**. Therefore, when developers were forced to provide system life cost estimations early on, which could turn out to be inaccurate, other **Cost Reducing Strategies** were required to be employed during the life of a system.

There are different system cost and effort estimation models. One example, is the COCOTS model (Abts, 2002; 2004), which provides estimations metrics for COTS component assessment, tailoring and glue code development. COCOTS does not however estimate the ongoing costs related to maintaining systems or take into account the communications overhead associated with system and software development teams, which was deemed to affect the productivity of development teams (Naunchan & Sutivong, 2007).

In their economic models, Lamine, Jalani and Ghezala (2006) and Lamine, Chouba and Bouzaida (2008) identified that the number of separate components used in a COTS-based system design, in particular Abts’ (2002; 2004) **CBS Functional Density Rule of Thumb**, should be taken into account when estimating the ongoing costs of COTS-based systems. Thus, Lamine et al. (2006) proposed that reducing the number of components
in a system, as recommended within the *CBS Functional Density Rule of Thumb* (Abts, 2002; 2004), could be seen as a *Cost Reducing Strategy*. They claimed that minimising the number of components reduced the number of interconnecting interfaces to be integrated and maintained, which in turn affected cost. Yang et al. (2005) suggested that the ideal *Cost Reducing Strategy* was to identify a single full-COTS solution, which satisfied all of the system requirements. These points were partly supported in the Grounded Theory analysis, whereby consciously applying the *Design Decision* to reduce the number of components included a system was seen as a *Strategy*. A factor not considered by Lamine et al. (2006), which emerged from the Grounded Theory analysis, was that the *reduced* number of components should all support the same architectural standard in order to reduce integration effort.

Furthermore, Lamine et al. (2006) identified that the *Implications Of Change* was a factor which could affect *Cost*. They argued that the maintenance cost of COTS components could be lower than those of other software items, as a result of vendors dividing the maintenance costs across their customer base. However, the cost of *Managing Change* of COTS-based systems, such as the re-tailoring effort incurred to replace components with upgraded components, could outweigh component maintenance cost savings, promoted by vendors. Thus, it was the effort required to change or replace components which was a major factor of core category *CONTROLLING COST*.

*CONTROLLING COST* also involved *Balancing Cost Challenges* relating to business opportunities, with those relating to business risks. A *balance* was between the sources of value gain and losses in system value (Boehm & Bhuta, 2008). Sources of value gain included improved COTS components and strategic partnerships. Losses in system value emerged from complications in vendor support, increased lifecycle costs and architectural mismatches between components. Boehm and Bhuta (2008) proposed the Incremental Commitment Model (ICM), whereby *Managing Change* was a key factor. Their model suggested that COTS-based system developers should be able to assess and manage opportunities and risks concurrently, rather than sequentially, in order to manage the rapid rates of change.

In relation to the Grounded Theory analysis the concept of *Managing Change* emerged from the data. *Change* was seen to influence *Cost* because it required effort. Change
required management because it could result in system instability or require the *redoing* of integration tasks etc. It can be seen that the ICM (Boehm & Bhuta, 2008) could add another concept to category **MANAGING CHANGE – Managing Risk**. Boehm and Bhuta (2008) proposed that risk management techniques should be employed to aid change management process whereby choices on whether risk avoidance, transfer, reduction or acceptance should be considered when changing COTS-based systems.

A *Cost Reducing Strategy* aimed at reducing software development costs was offshore outsourcing. This tended to involve moving development activities to low-cost development environments that are locally managed (Islam, Joarder & Houmb, 2009). In the domain of CBD, offshore outsourcing has been used for the production of custom and glue code for system developers and the development of components for vendors. However, potential problems with offshore outsourcing occurred as a result of the control and responsibility for software development tasks being moved away from the clients. This feature of offshore outsourcing was considered to increase the chance of project failure. Thus, choosing offshore outsourcing required a trade-off between reducing development costs and loss of control (Islam, et al., 2009). Loss of control can result in increasing cost when problems occur.

### 6.2.2 Design Principles

An objective of system **DESIGN PRINCIPLES** was to support the selection of COTS components, to support system requirements, and to facilitate their integration with minimal effort. System requirements included the following:

- **Business requirements:** the company’s goals;
- **User requirements:** the tasks users need to be able to achieve;
- **Functional requirements:** Functions required to be built into a system;
- **Business rules:** Organisational policies, standards and rules which must be incorporated into a system (Finnegan, 2008).

A consequence of selecting components which individually provided the required functionality but which were incompatible when combined was *increasing* development and maintenance costs as a result of the additional tailoring and integration effort (Zhou, Cheng, Zhou, Li & Hu, 2008). System designers may be forced into **Balancing Design**
Principles by selecting components which do not supply all of the required functionality but which are compatible. This was supported in the Grounded Theory analysis. It was suggested that there could be a fine balance between identifying compatible components, which also supported enough of the system requirements to justify their selection.

Yang et al. (2005) suggested that Conflicting Design Principles could result from using unsuitable design models such as the waterfall model to produce COTS-based systems. System designers could not guarantee that components which supported the system requirements would be available.

Precursors to selecting components required the evaluation of candidate components (Shyur, 2006; Sheng & Wang, 2008), as well as an understanding of the system requirements (Finnegan, 2008). A challenge for system designers was to decide exactly what type of system to build (Brooks, 2002) as there was often a delta between available components and system requirements.

To evaluate prospective components the following criteria should be considered:

- functionality;
- non-functional attributes;
- architectural compatibility;
- business considerations (Land, Blankers, Chaudron & Crnkovic, 2008).

It was recommended to evaluate combinations of components early on in order to identify and address architectural mismatches. In conjunction, ‘keystone technologies’, which includes suitable operating system platforms on which to host COTS-based systems, should be also identified early on in the evaluation process as this may exclude many components (Land et al., 2008).

In many cases, there was a gap between COTS features and system requirements (Sheng & Wang, 2008; Suleiman, 2008). Selecting components from a set of potential alternatives could be a complex process involving the consideration of Conflicting Design Decisions and Conflicting Design Principles. Furthermore, interdependencies and incompatibilities existing between components can make the evaluation of a
component in isolation of other components difficult. Thus, determining the *Degree Of Dependency* between components was deemed key. Furthermore, Bhuta and Boehm (2007) suggested that performing interoperability assessments between components became further complicated as the number of components increased. From the Grounded Theory analysis *Degree Of Dependency* emerged in relation to the maintenance of systems. However, it can be seen that *Degree Of Dependency* also applies to the development of systems as the dependency between components and other system parts when systems are being developed.

Yang et al. (2005) warned that letting system requirements drive COTS-based system production should be avoided because this reduces the ability for system developers and maintainers to control system lifecycle costs. This statement was made in relation to the use of the waterfall software design model, which was claimed by Yang et al. (2005) to be unsuitable for COTS-based system development. The reason given was that system requirements, which were agreed early on in a project, tended to be viewed as non-negotiable by customers and associated with fixed priced projects.

The Analytic Hierarchy Process (AHP) was developed as a method to evaluate the suitability of components (Shyur, 2006). The benefit of AHP was that it was designed to provide an understanding of the complexities in areas involving multiple criteria, which included the interdependencies between components.

Evaluation of components should be made within the context of the system in order to determine how they work when integrated together; the quality of a component may appear different when assessed in isolation (Suleiman, 2008).

Lee, Lee, Oh and Wu (2008) suggested that the ability to understand *System Complexity* presented a major challenge for COTS-based system developers and maintainers because of the affect on system stability of changing system parts. The trend has been for software systems to become more complex. Systems are required to perform greater numbers of different computations. The Bayesian Belief Network (BBN) approach was proposed to aid the ability to evaluate and test components in complex environments. Bayesian learning incorporates inductive inference based on past experiences, enabling the dependency relationships between system parts to be represented. This was deemed to be useful approach to understand system complexities and dependencies in domains.
where prior data existed. However, it was not possible to determine the accuracy of BBN when analysing components in new environments (Lee et al., 2008).

The use of gap analysis was proposed as a suitable method to evaluate the suitability of COTS components (Sheng & Wang, 2008). This was a requirements-driven approach. As such, the claimed benefits were that gap analysis was able to determine the capabilities COTS components lacked. Both functional and non-functional gaps could be identified and a weighting assigned to represent the degree of difficulty in modifying components.

An option which enabled system designers to deal with *Conflicting Design Principles*, which resulted from the incompatibilities between available components, was Aspect-Oriented Software development. This approach involves developing integration *aspects* to wrap COTS components. The claimed benefit was that creating *aspects* enabled previously incompatible components suitable to be integrated into the same system (Perez et al., 2008). However, no details were provided on the amount of effort required to produce *aspect* wrappers or the effort required to upgrade or patch *aspect* wrapped components.

It was suggested that there should be architectural conformity between all aspects of a system including the hardware, COTS-components and other system parts. Tailoring components to make them function within different architectural standards was seen to be very time consuming and as such, contributed to cost (Suleiman, 2008). Thus, *Design Decisions* leading to the selection of components with shared characteristics was considered beneficial (Gamble & Gamble, 2007).

### 6.2.3 Organisational Issues

In an early operations management article Lewis (2003) suggested that the impact and nature of organisational risks were not fully understood. *Supply chain management*, which was defined in terms of the cost, quality and confidence in the supply of products, was identified as one area of concern. It can be seen that for COTS-based systems the supply of reliable, quality products, at the right cost is key, as is the confidence in a supplier’s ability to support the products. With reference to the Grounded Theory
Analysis, system developers’ confidence in supply chain management relates to ORGANISATIONAL ISSUES: system developers are reliant on the cooperation of vendor organisations to supply, maintain and support components. Furthermore, the effect of supply chain management issues, such as the supply of poor quality components, may be Beyond The Sphere Of Influence of system developers who tend to lack control over vendor organisations.

In more recent articles, organisational risks have been identified as being significant contributors to Cost, when one organisation is reliant on the cooperation and products provided by other organisations (Kull & Talluri, 2008). In the domain of COTS-based system design system designers and developers are reliant on the support and quality of components supplied by component vendors.

Kull and Talluri (2008) identified the following sources of supply risk:

- Delivery failure: A supplier fails to make a delivery when promised;
- Cost failure: The price of the supplied product rises above expectations;
- Flexibility failure: The supplier refuses to make design changes;
- Confidence failure: A supplier drops in standing as a reliable supplier.

Although Kull and Talluri (2008) identified these risks from case studies involving manufacturing companies it can be seen that the similar risks can affect COTS-based system developers. For example, on Delivery failure COTS component vendors can fail to provide components within the agreed time frame. Vendors may also fail to resolve software bugs in a timely manner. Kull and Talluri (2008) explained that Delivery failure can occur as a consequence of suppliers reaching full capacity or as a result of unreliable raw material sources. In the COTS domain unreliable raw material sources could be equated to vendors not having the personnel with the required levels of skills.

On Cost failure component vendors can hold a degree of power in the COTS market place allowing them to dictate the price of their components. Component customers may be powerless to influence this when the components are integral to the software systems.
Quality failure of COTS components can occur when vendors do not have the ability to control the quality of their products or where they are unconcerned with product quality and customer satisfaction. COTS vendors and other types of suppliers can get away with this when they hold the market share of the products.

Flexibility failures can occur in the COTS market place when vendors are unable or unwilling to modify components or resolve software problems for individual customers. A reason for this is that vendors tend to produce identical copies of components for sale within the marketplace. When individual customers request additional features to be included or specific bugs to be fixed, which are not experienced by other customers, vendors are more likely to wait and include these in product refreshes.

Confidence failures in customers’ perceptions of vendor organisations can develop as a result of vendors producing poor quality components or failing to provide adequate product support. Confidence failures can occur as a consequence of vendors failing to manage the production and support of products and fail to develop working relationships with their customers (Kull & Talluri, 2008).

With reference to the Grounded Theory analysis it can be seen that the ability of customers (system developers) to affect delivery, cost, quality and flexibility issues within vendor organisations may be Beyond Their Sphere Of Influence. Customer organisations do not normally have the ability to influence a vendor’s working practices. Furthermore, when only one vendor produces the required component or where customers have invested a lot of time and money integrating components supplied by a vendor customers may not be in a position to find alternative product sources.

Supplier failures can be detrimental to a customer’s business when customers are reliant on a supplier’s products (Kull & Talluri, 2008). Therefore, it was deemed important that system developers evaluated strategic suppliers before embarking on using their components. Kull and Talluri (2008) developed a decision model to aid strategic supplier selection. It focused on the supplier failure likelihood and the impact of a supplier failure. However, a problem with this model was that supplier confidence levels were difficult to assess. Furthermore, the model assumed that current supplier capabilities will remain constant over time. Some suppliers may be actively endeavouring to improve their performances. This would not be captured in the model.
From a people management perspective effective ways of leading people can differ depending upon the national CULTURAL ISSUES (Hofstede, 1994). This had implications for large COTS-based system development projects where development and maintenance project teams were located in different countries. Project teams in one country may work differently and offer better levels of cooperation compared with teams from other countries. Therefore, Organisational Concerns may not just relate to system developers concerns over vendor organisations but could also relate to concerns over other parts of system developer’s own organisations.

Organisational culture can affect the way people behave within organisations. Ankrah, Proverbs and Debrah (2009) stated that were different facets to organisational culture, which included characteristics of the industry organisations operated in and sub-cultures expressed by different groups of people within organisations. It was possible for sub-cultures to exist within an organisation whereby the goals of individual employees did not match the goals of the organisation (Ankrah et al., 2009). It was seen from the Grounded Theory analysis that the Support Quality provided by vendors varied between vendor organisations and between individuals within support teams. However, it can be suggested that Support Quality would diminish when the culture of organisations do not promote the ideals of good support quality and customer care. Thus, the effect of the organisational culture of vendor organisations and support teams may be Beyond The Sphere Of Influence of system developers.

Furthermore, with reference to the Grounded Theory analysis it can be seen that Conway’s Law (Brooks, 2002) adds another property to category ORGANISTATIONAL ISSUES; the concept Organisational Structure. The basis of Conway’s Law is that the structure of bespoke IT systems tends to match the organisational structure of an organisation. However, because COTS components tend to be produced by vendor organisations, rather than by the organisation developing the system, it can be suggested that Conway’s Law will not apply to COTS-based systems. Thus, with this premise in mind the implementation of COTS-based systems is more likely be met with Resisting Change by users, if they are forced to change their working practices to match the functionality of a system which does not necessarily match their organisational structure. However, further research is required into this area.
6.2.4 Managing Change

A factor affecting computer systems is the rapid pace of change in both technology and business focus. Thus, in today’s business environments systems are required to continuously evolve to address new business opportunities and embrace emerging technologies. Hence, concepts Managing Change and Designing For Change constitute major concerns of system designers and developers. It can be seen that here is limited benefit of designers producing inflexible systems because later on in the system lifecycle these systems will invariably need to be changed.

Therefore, for COTS-based systems, flexible architectural designs should be used in order to accommodate changes occurring due to the volatility of the COTS marketplace and component upgrades (Suleiman, 2008).

Yang et al. (2005) indicated that Designing For Change was an essential precursor to enable system developers and maintainers to Manage Change. Yang et al. (2005) stated that, on average, COTS vendors upgraded their components every ten months. Version releases, on average, went out of support after three releases. Furthermore, there was no guarantee that all vendors followed the same upgrade cycles. Therefore, the effort involved with Managing Change was likely to increase with the greater number of COTS components used in a system.

6.2.5 Cultural Issues

The quality of the technical documentation relating to COTS components can vary. In many cases this documentation poorly describes the functionality of products (Trivedi, 2007; Chang et al., 2009). The assumption is that this can occur as a result of CULTURAL ISSUES where component vendors, technical authors and components buyers are Lacking Common Understanding of the language and terminology used to document product functionality and usage.

With reference to the Grounded Theory analysis it can be seen that the acquisition of inappropriate components could occur as a result of inaccurate or misleading technical specifications created to describe the functionality of components. Integration problems
can occur as a result of acquiring inappropriate or incompatible components. Furthermore, the use of incompatible components can conflict with accepted **DESIGN PRINCIPLES**. Thus, **CULTURAL ISSUES** can be an influencing factor of the application of **DESIGN PRINCIPLES**.

The Grounded Theory analysis identified **CULTURAL ISSUES** which related to the misinterpretation of language. However, the affect of culture on system development may be much greater than language related differences. This is an area for further focus because cultural influences now play increasingly important parts within business relationships, as a result of the increasing globalisation of business. Businesses are now more willing to utilise the reduced labour costs of people from countries, such as, China, India and South America. Misunderstandings can arise in international trade due to difference in cultural background of trade partners (Hofstede, Jonker, Meijer & Verwaart, 2006). This is important where the trust and the relationship between partners are deemed important. For COTS-based design cultural influences can affect the relationships between system designers and developers and vendor organisations. If relationships become strained the ability for system developers to successfully **CONTROL COSTS** can be challenged because system development tasks require human interaction and cooperation between different parties. Thus, to prevent additional costs resulting from the consequence of problems system development tasks need to run smoothly.

Hofstede (1994, p. 4) defined culture as, “the collective programming of the mind which distinguishes the members of one group or category of people from another.” Thus, culture represents the beliefs, values, norms, practices, meanings and symbols that people develop, share and learn as members of a certain group. Culture applies to nations, organisations, professions, age groups etc. However, cultural differences are most clearly recognisable at national levels (Hofstede, 1994). This can be explained by the notion that national beliefs and values of a country can affect all people and permeate through organisations and other groups. The implications are that the way organisations operate can be affected by the national culture of their country of origin.

Culture is what distinguishes one group of people from others. Thus, culture is not an attribute of individual people, but an attribute of a group that manifests itself through the behaviours of its members (Hofstede et al., 2006). Hence, from the perspective of
COTS-based development cultural aspects may affect the quality of support vendor organisations give to their customers. For example, one generalisation made about people in India by those in Western Europe is that Indian people do not like admitting to problems occurring. Therefore, vendors based in India are less likely to openly acknowledge problems occurring with their products. In Japan, a cultural pattern is for Japanese people not to say ‘no’. Therefore, as a result they are likely to avoid discussing areas where they cannot provide the expected level of support or where a product is not functioning as expected. Their likely response is to discuss what they can do or where the product is working correctly.

Some of the recognised management theories cannot be applied to all national cultures (Hofstede, 1994). For example, Hofstede (1994) stated that McGregor’s Theory X-Theory Y becomes irrelevant in Southeast Asian countries because of different culturally determined assumptions. For example, in the USA and Western Europe the following national assumptions are made:

- Work is good for people;
- People’s capacities should be maximally utilised;
- There are organisational objectives that exist in isolation from people;
- People in organisations behave as unattached individuals.

These differ in Southeast Asian countries:

- Work is a necessity, but not a goal in itself;
- People should find their rightful place, in peace and harmony with their environment;
- Absolute objectives exist only with God. In the world, persons in authority positions represent God, so their objectives should be followed;
- People behave as members of a family and/or group. Those who do not are rejected by society (Hofstede, 1994).

With reference to Hofstede (1994) and the Grounded Theory analysis it can be seen that conflicts may exist between COTS-based system developers and COTS component vendors if both parties express different national cultural characteristics. For example, if
a vendor organisation originates from a culture where support quality is dependent upon environmental harmony or religious ideals then system developers based in Western cultures may not receive the expected level of product support.

Furthermore, cultural aspects can add additional dimensions to concepts, such as *Conflicting Business Motives*, which emerged from the Grounded Theory analysis. Concept *Conflicting Business Motives* related to the notion that system developers aim to source the most suitable components, underpinned by good product support, at the lowest cost in order to satisfy the perceived cost savings of building systems from components. On the other hand, vendors aim to develop commercially viable components. The interview data suggested that both system developers and vendors normally shared similar cultural values. However, if system developer organisations originate from cultures which are different from those of the vendor organisations, *Conflicting Business Motives* may encompass factors other than issues of financial profit.

Culture at the national level is concerned with five big issues, or dimensions of social life: hierarchy, identity, cooperation-performance orientation, the unknown and the gratification of needs (Hofstede et al., 2006). Hofstede and Hofstede (2005) conceptualised each of these dimensions as a bipolar continuum ranging from 0 through to 100, which are as follows:

- Hierarchy defined from small to large power distance;
- Identity from collectivist to individualist;
- Cooperation-performance orientation from cooperation oriented to performance oriented;
- The unknown from weak to strong uncertainty avoidance;
- Gratification of needs defined from short-term to long-term orientation.

Hofstede et al. (2006) identified how extremes of, for example, the ‘cooperation-performance orientation’ dimension could manifest itself in reality:

- Performance oriented cultures value measurable performance criteria, such as size, speed and quantity. Wealth is admired. Life is conceptualised as a series
of contests where winning is fundamental. Losing is a disaster. Implicit trust is low, thus when someone is cheated it is seen as their own fault. Big is admired in all fields.

- Cooperation oriented cultures are the opposite. Winners can cause feelings of jealousy. Implicit trust is high and those flouting rules are disliked. Good intentions are more important than good performance.

It should be stated, however, that behaviour is a mix of dimensions originating cultural and personality influences. Culture is multi-faceted. The basis of the Hofestede’s cultural dimensions is unclear as there is no suggestion that dissecting peoples brains would reveal these dimensions. As a result, Hofestede has been criticised on the basis of these distinctions. In contrast, Hofestede argued that the distinctions originated from mathematical and statistical analysis of data collected from his research. As such, the dimensions should be viewed as descriptive, not prescriptive or causal models (Biddle & Dormann, 2009).

Cultural differences can be wider than mere differences in national culture. People are now much more mobile and are more likely to work in different countries. Therefore, people can carry around different cultures, such as their home culture and culture of their country of work, as well as switching between different cultural dimensions between work and home (Biddle & Dormann, 2009).

In the context of COTS-based development it can be seen that cultural dimensions can impact the ability of system designers, developers and maintainers to CONTROL COSTS. For example, where a customer of a COTS-based system is performance oriented their main goal may be for the system to be developed as quickly as possible with the greatest cost savings. Conflicts may occur if the system developer is not from the same performance oriented culture.

Cultural dimensions can also affect vendor organisations and the quality of support they provide for their components. Performance oriented vendors may be less willing to spend additional time, and thus incur more costs, providing support to individual customer’s problems. Conversely, cooperation-oriented vendors may be more likely to
provide a better level of product support because their aim to provide good service is part of their cultural make-up.

Project dependent factors have been identified to influence organisational culture, which also influence the performance of project teams and thus, the cost to deliver projects. Project dependent factors include: the nature of the dominant participants who drive projects, the goals and objectives of a project and a project’s location (Ankrah et al., 2009).

The effect of **CULTURAL ISSUES** on system developers may not be obvious. For example, off-shoring can be integral to a company’s strategic plan (Trivedi, 2007). Vendor organisations may offshore component development and product support operations to other countries - but they may not publicise this to their customers. Thus, system developers may source components from vendors believing that the products are produced and supported from the same country as the vendor, when in fact the products are produced in countries where support quality is deemed poor by system developers.

Cultural differences may become more noticeable as a result of the popularity for companies to implement strategic offshoring and to distribute development teams across different countries (Woodward et al., 2010). Cultural differences can impact the effectiveness of distributed teams to communicate and collaborate. For example, in some cultures it is not normal for a person to admit that they do not understand other team members. In other cultures it can be usual for someone to say ‘yes’ to a statement without ever challenging or questioning the content (Woodward et al., 2010). Thus, in such situations misunderstandings were likely to occur impacting the productivity of the team.

Woodward et al. (2010) also stated that language differences impacted the ability for people to communicate effectively with distributed teams. Language challenges appear in several forms. For example, distributed team members may not have the same first language, some members may not speak the language used by the majority of the team and as a result of different accents some team members may not be able to understand other members. **CULTURAL ISSUES** which related to language differences emerged from the Grounded Theory analysis. Concepts *Lacking Common Understanding* and *Cultural Misunderstandings* conceptualised the challenges experienced by system
developers when communicating both verbally and in the written form to people from different countries.

6.2.6 Managing Complexity

The complexities of COTS-based systems have grown immensely over recent years with an increasing interoperation of wider varieties of components, architectural standards and technologies (Oberndorf & Kark, 2007). The challenges involve system developers managing these complexities in order to develop, deploy and sustain these systems.

One method of managing the complexities of integrating components which are not compatible with each other involves the creation of integration code to improve the compatibility of components. However, the cost and effort of producing integration code can be high. The costs of developing integration code can increase dramatically as the number of components used in a system increases (Egyed & Balzer, 2006). With reference to the Grounded Theory analysis it can be seen that MANAGING COMPLEXITY of systems, as a way of CONTROLLING COST, can involve selecting fewer components, thus requiring the production less costly integration code.

6.3 Conclusions

The purpose of this literature review was to examine the literature to ascertain if there was support for the results of the Grounded Theory analysis. The core category, CONTROLLING COST, was implied within the literature, because the choice to use COTS-based development methods in the first place was normally made in relation to a perceived cost saving. Cost saving was assumed to result from reducing system development effort or increasing system flexibility.

The literature reviewed indicated that system development and maintenance Costs were affected by different issues. These included issues relating to Design, Complexity, Change, Organisation and Culture.
An issue which was dealt with more fully in the literature than within the Grounded Theory analysis was the effect of **CULTURAL ISSUES** on *Cost*. COTS-based system development, as a human task, requires the collaboration of different groups of people, ranging from system designers, project managers, vendors, support teams, customers etc. With an increasing focus on offshore outsourcing there may now be a greater requirement for collaboration across organisational and cultural boundaries. Hence, projects conducted in multicultural settings, present many challenges, ranging from project planning and management, collaboration and language interpretation.

The impact of *Cultural issues and differences* between people on *Cost* and COTS-based system development methods is an area which has been identified for further investigation.

The next chapter provides a reflection on the experience of using the Grounded Theory method within the domain of CBD.
Chapter 7 Reflections on using the Grounded Theory Method

7.1 Introduction

The Grounded Theory method was the chosen method to analyse data provided by IBM practitioners involved in CBD because it was claimed that this method could enable the generation of a theory to explain an area of concern (Glaser 1978; Strauss & Corbin, 1998; Charmaz, 2006). In the case of this study Grounded Theory might have generated a theory to explain the results of the statistical analysis part of the study.

The aim of this chapter is to reflect upon the experience of using Grounded Theory and to establish if it was an appropriate method to use within this study.

7.2 Justifications for Using Grounded Theory

When looking for candidate methods to explain the statistical analysis results, a theory ‘creation’ method, such as Grounded Theory, was deemed a more relevant choice to achieve this rather than a theory ‘testing’ method because at this stage it was not possible to offer a theoretical explanation of the statistical analysis results. Furthermore, because the results of the statistical analysis did not support the CBS Functional Density Rule of Thumb (Abts, 2002; 2004) it also seemed appropriate to select a method which might explain the reason for these results.

However, other qualitative data analysis methods that claimed to be suitable for explaining research areas were considered. For example, the Explanatory Effects Matrix (Miles & Huberman, 1994) was considered. It was claimed that this method could assist in providing causal explanations within an area of study. This was achieved by building up a matrix, where each row represented one theme from the data. One column in the matrix was allocated to ‘researcher interpretation’. It was here that the researcher entered an interpretation of each theme. However, the method did not prescribe how this was to be achieved. For example, it was not clear if interpretation should just consist of a description or a summary of the theme or whether it should focus on a conceptual explanation of the theme. This method was rejected for this study because as the number of rows increased within the matrix details on how to gain an understanding of the complexities between the themes contained in each row were not made clear.
However, Lehmann (2010) claimed that what differentiated Grounded Theory from other qualitative methods were the rigor of the analysis regime, the basis of which was to collect and analyse data together and the use of the constant comparison method. It was the perceived rigor of the Grounded Theory method which was a contributing factor for choosing Grounded Theory for this study.

7.3 Choice of which Grounded Theory Method to Use

As explained in Section 3.6.3 there are various flavours of the Grounded Theory method. However, for this study the original aim was to base the analysis on Glaser’s (1978) Grounded Theory method because of a number of reasons:

Glaser (1978) stated that the power of theories emanating from the Grounded Theory method were that they should be relevant to other areas of concern and not remain tied to the areas of study. He suggested that this could only be achieved by ensuring that during the analysis process codes, concepts and categories remained abstract of time, place, people and context. This was an appealing feature for this study because it was felt that a theory explaining the cost factors of building systems from COTS software components would have a greater significance if it could be generalised to other domains where solutions were developed from off-the-shelf type entities.

Glaser (1978) recommended performing *Theoretical Coding*, in which relationships between categories are attributed, later on in the Grounded Theory process once all of the categories had emerged. This seemed to be a logical approach to take because at that stage a fuller picture of the analysis would be available as all constituent categories had been identified. Strauss and Corbin (1998), on the other hand, suggested performing an equivalent of theoretical coding, *Axial Coding*, early on in the analysis to attribute links between categories as soon as they emerged. However, it was felt that performing this type of coding at this stage of the process was too early, because the time spent may be wasted attributing associations between categories when the focus may change if other categories emerged.
However, for this study a hybrid Grounded Theory approach evolved because it was not possible to adhere directly to Glaser’s (1978) method. For example, Glaser (1978) stated that a Grounded Theory study should commence before performing a literature review or defining the problem area. His justification of this was to prevent researchers from starting the research with preconceived ideas of the areas of concern. However, the choice to use Grounded Theory was made later on in this study, in order to explain the results of the statistical analysis. By this time the literature review had already been performed and a focus for the study identified; Abts’ CBS Functional Density Rule of Thumb (2002; 2004).

Another reason for adopting a hybrid Grounded Theory approach concerned the limited instructions on how to perform conceptualisation, an integral part of Glaser’s (1978, 2001) Grounded Theory method. Glaser (2001, p. 2) stated that the difference between his Grounded Theory method and other qualitative data analysis methods was that his “exists at a conceptual level and is composed of integrated hypotheses” whilst other qualitative data analysis methods merely “produce description with or without conceptual description mixed in” Glaser (2001, p. 2). His justification for the focus on conceptualisation was to ensure that the analysis remained abstract of time, place and people, in order to be able to generalise the results within different areas of concern. However, on the requirement for researchers to perform conceptualisation correctly Glaser (2001) was strict. He stated that researchers without the ability to conceptualise should not use Grounded Theory but should revert to using other data analysis methods where description was acceptable.

7.4 Experience of Using Grounded Theory in this Study

During this study much time was spent attempting to gain an understanding of the art of conceptualisation, as prescribed by Glaser (1978; 2001). Questions such as ‘is this being conceptual?’ came to mind many times during the coding and analysis process. In one example the following advice was received from the researcher’s director of studies, Dr Allan, in answer to the question ‘Is this Memo getting more conceptual?’, following a discussion of a memo written early on in the analysis aimed at explaining the relationships between concepts ‘Continuing’ and ‘Minimising.’
“You have arrived (somewhere else) at the concept of continuing (similarly for minimising etc. and all of the others) and then you go and describe what they mean. This defeats the intention of conceptualising to get away from description of what it means. Having conceptualised you should be able to define the concept yourself from its emergence from the data - I suggest you try this using the method we discussed in my office - that is generalising away from specifics in the data. Try it and see what you come up with. This will probably need lots of practice and be difficult at first but do not shy away from it because it is hard. Get practising and overcome the hardship. Later on you discuss possible links and non-links - this is good and will lead to clarity eventually - so again my message is ‘stick with it - do more of it’. Get away from describing whether there might be a link or there might not. Look for links, imagine links and try links to see if they are ridiculous” (personal communication, March 14, 2007).

However, with reference to the above example, during the analysis it was felt that description formed a necessary part of expressing the nature of codes, concepts and categories and was required as part of the process of developing the relationships between these entities through the writing of memos. By endeavouring to adhere to Glaser’s (1978; 2001) requirements during the analysis process it was a challenge to decide on when description should end and when ‘conceptualising’ should commence.

Van Niekerk and Roode (2009, p. 98) suggested that experiencing difficulty with applying Glaser’s Grounded Theory method, especially for researchers new to Grounded Theory, was not a new problem because “very few people have received training in conceptualisation.” However, Glaser (2001) justified not making conceptualisation and other facets of his method more prescriptive because he argued that this would restrict the method considerably. The claimed strength of Glaser’s method lies not only in its rigour of following systematic rules to derive a theory, but also in its flexibility and creativity when conceptualisation needs to take place (Van Niekerk & Roode, 2009).

Strauss and Corbin’s (1998) Grounded Theory method advocated a staged approach to generating a Grounded Theory, which moved from description, conceptual ordering through to theorising. They acknowledged Glaser’s (1978) point that with greater abstraction of concepts the broader their applicability, but warned that with too much
abstraction the concepts would be further removed from the data they pertained making linking the resultant theory back to the study area difficult.

In hindsight, for this study incorporating the prescriptive aspects of Strauss and Corbin’s (1998) Grounded Theory method into the hybrid method may have been more suitable for dealing with the data analysis. It is now felt that starting with description, before moving to conceptualisation, would have assisted in exploring the essence of concepts and the discussion of relationships between codes, concepts and categories within the memos. Adolph, Hall and Kruchten (2008) suggested that the prescriptive approach of the Straussian Grounded Theory method has appealed more to researchers from IS domains because such researchers tended not to be trained in social science research methods. From the research papers Adolph et al. (2008) surveyed, which claimed to follow the Grounded Theory method, more appeared to follow the Straussian approach, rather than Glaser’s Grounded Theory method.

A dilemma of Glaser’s Grounded Theory method concerned theoretical coding. Glaser (1978) supplied a set of 18 coding families, to be used as the basis for explaining the relationships between concepts and categories. However, other authors (Strauss & Corbin, 1998; Charmaz, 2006) suggested that Glaser’s coding families constituted the introduction of ‘pre-conceived’ codes, which in a method which rejected the use of pre-conceived ideas was contradictory. However, Glaser’s (2005) argument was that researchers may create their own theoretical codes in situations where his coding families were not suitable.

For the purpose of this study the use of Glaser’s (1978) coding families proved useful and provided a starting block from which to identify relationships between concepts and categories. For example, the Six C’s coding family was used most often in the analysis as the relationships between many of the concepts encompassed a causal/consequence model. The popularity of the Six C’s may be explained by the view that this coding family is the “bread and butter theoretical code” (Glaser, 1978, p. 74). However, during the analysis there was an assumption that at least one of the coding families must fit each relationship. Therefore, researchers may end up forcing a code onto a relationship when it is not wholly appropriate. This may be a weakness of Glaser’s (1978) theoretical codes.
7.5 Different Interpretations of Grounded Theory

A problem was encountered defining the nature of a *Grounded Theory* in this study. It was clear from the literature that not all authors in the Grounded Theory field viewed a *Grounded Theory* in the same way. For example, Glaser’s Grounded Theory method was based on the positivist definition of theory whereby the emphasis was on cause, deterministic explanation and generality (Glaser, 1978; Charmaz, 2006).

Alternatively, Strauss and Corbin’s (1998) Grounded Theory method included an interpretive definition of theory which also emphasised understanding, rather than just explanation. This view of Grounded Theory required researchers to interpret data in order to gain an understanding of the data. Additionally, interpretive theory development required the imaginative understanding of the study area (Charmaz, 2006), which suggested that one researcher’s interpretation of the data may differ from others. However, Strauss and Corbin’s (1998) Grounded Theory method included some positivist aspects of theory as well as emphasising relationships between concepts. A grounded theory was defined as “a set of well-developed concepts related through statements of relationship, which together constitute an integrated framework that can be used to understand, explain or predict phenomena” (Strauss & Corbin, 1998, p. 15). The ability to generalise the grounded theory to other areas was less of a focus of this method.

Charmaz’s (2006) Grounded Theory flavour adopted the constructivist approach to theory generation. She explained that this placed priority on the phenomena of a study, viewing both data and analysis as being created from shared experiences and relationships with participants and other sources of data. Thus, the resultant Grounded Theory would be influenced by the researcher’s views.

The above points highlight potential issues with the Grounded Theory method, whereby the end result may vary depending upon the flavour of Grounded Theory used and the researcher performing the analysis. For example, with Strauss and Corbin’s (1998) and Charmaz’s (2006) Grounded Theory methods, where theory generation relies on researcher interpretation, there may be a greater chance that resultant theories will differ as a result of different researchers’ interpretations of the data. However, as mentioned
above, the original aim in this study was to adhere to Glaser’s Grounded Theory method because of his claim that *conceptualisation* added validity to emerging concepts, categories and the theory itself. However, for this study and in the absence of clear instructions on how to perform conceptualisation (Van Niekerk & Roode, 2009) it seemed clear that the process of conceptualisation required a degree of interpretation by the researcher in order to link a pattern of events to a concept. For example, when considering body language, deciding if an interviewee is being sincere or anxious etc. may be more open to researcher interpretation.

However, it could be suggested that the results from other qualitative data analysis methods involving the *interpretation* of events may also suffer from a variation between different researchers’ interpretations. For example, with *Content Analysis* (Miles & Huberman, 1994; Coolican, 2009) counting the number of times an interviewee mentioned a particular topic may be a straightforward operation. However, in order to come to some conclusion regarding the implications of the number of mentions of a topic the analysis would still require interpretation by the researcher.

### 7.6 Suitability of the Grounded Theory Method for this Study

The question posed by the outcome of the Grounded Theory study is why this Grounded Theory study did not result in a neat novel theory which explained the statistical analysis results and/or which explained the cost factors of CBD.

In hindsight, an answer to this question may be related to the researcher failing to fully appreciate the nature of the differences between the Grounded Theory flavours when planning the study. For example, according to Van Niekerk and Roode (2009) Glaser’s and Strauss and Corbin’s Grounded Theory methods *should not* be mixed because they are substantially different and not reconcilable. A Glaserian Grounded Theory study, for example, should not start with a research question or a literature review to avoid forcing pre-conceived ideas on the study. Conversely, the researcher starts with a literature review and research question with Strauss and Corbin’s method because this is claimed to guide the research. Thus, with reference to Van Niekerk and Roode (2009) it can now be suggested that a hybrid Grounded Theory approach involving only part of Glaser’s Grounded Theory method was inappropriate and would not have resulted in the
emergence of a Grounded Theory. The reasons for this were that for this study the literature review was conducted at the start of the research project and the focus of the Grounded Theory part of the study, which was to develop a theory to explain the results of the statistical analysis, was pre-selected, thus resulting in the pre-conception of the problem area from the start. Additionally, the decision to focus on the *CBS Functional Density Rule of Thumb* (Abts, 2002; 2004) early on may have contributed to forcing the data collection and analysis focus, further reducing the chance of a new theory from emerging.

Furthermore, it was not possible to explicitly link practitioners’ comments to the specific projects from which the data used in the statistical analysis represented because further knowledge about the projects was not available. However, because the data represented projects developed solely within IBM it may be suggested that the practitioners may be implicitly linked to the projects because of their length of service and experience of implementing COTS-based systems within the company. However, in this study these factors were not deemed to preclude the use of the Grounded Theory method. The use of any other method to analyse practitioner comments would encounter the same limitations.

Another answer to the question of why a Grounded Theory did not emerge may be related to the scope of the study. The underlying scope of cost was large. It seemed clear from the analysis that there were many different CBD cost factors. This was seen in the complexity of the relationships between the concepts and categories which emerged from the Grounded Theory analysis. Thus, the *Controlling Cost* core category (see Section 5.8.1) was dependant upon issues relating to *Managing Change, Managing Complexity, Design Principles* and *Organisational and Cultural Issues*. A theory which adequately captured the underlying complexities of each category, as well as the complexities of the relationships between categories did not seem to emerge.

Therefore, with reference to Strauss and Corbin’s (1998) basis of theory, whereby the aim of a Grounded Theory is to understand, explain or predict, for this study it seemed more practical to articulate the ‘Grounded Theory’ as a set of ‘Grounded Principles’ (see Section 5.10), each of which could contribute to an understanding, explanation or prediction of areas of concern. The source of each ‘Grounded Principle’ could be linked
back to relationships between concepts and categories explored within the earlier Grounded Theory analysis.

Different results may have been produced by reducing the scope of the study to focus solely on one cost factor (or to conduct more than one study focusing on different cost factors), for example, Managing Complexity. However, this approach could have detracted from the fuller understanding of the factors affecting the costs of CBD. It seemed clear that multiple factors appeared to influence the cost of the CBD approach.

An alternative answer to the question of why a Grounded Theory did not emanate from the analysis concerned the nature and knowledge of the practitioners interviewed for the Grounded Theory analysis. The average time they have worked for IBM was 11 years. The assumption was that given their position within the company – Architect, Project Manager and Developer – that they would have experienced a high degree of education in all facets of system development theory and methods and thus, possessed a level of commonly accepted knowledge. Thus, it could be suggested that their knowledge of CBD theory biased their views. Lehmann (2010, p. 1) stated that the Grounded Theory method aims to “create theory in domains where there are none”. Therefore, with reference to Lehmann (2010) it may not have been possible to use Grounded Theory to generate a novel theory to explain an area of concern when theories already existed and where all of the participants were familiar with the existing underlying theory of system development. Thus, for example, if the participants assumed the CBS Functional Density Rule of Thumb (Abts, 2002; 2004) to be true and were not open-minded enough to embrace new ideas then it could be suggested that the Grounded Theory method would be unlikely to result in an alternative view despite the statistical data showing something different. However, with this in mind it would be unlikely that other explanatory analysis methods, which relied upon the views of participants, would produce novel theories either in this situation.

However, on this point Glaser (2001) stated that the Grounded Theory researcher must not let the participants of Grounded Theory analysis force theory generation. He suggested that participants are merely the source of data and that it is through the skill of a proficient Grounded Theory researcher that something new will emerge from the analysis. However, in this study, if interviewees relayed their beliefs on the merits of
particular software development approaches, which were related to their experiences or education, it would have been difficult to attribute alternative meanings.

This poses the question as to whether Grounded Theory (or the hybrid Grounded Theory method used in this study) was a suitable method for investigating areas of concern where the participants were familiar with existing theory relating to the area of concern. The origins of Grounded Theory were in the field of sociology whereby Grounded Theory was developed and used to explain the complexities of social settings where it could be suggested that participants of such studies were not necessarily versed in sociological theory (Glaser, 1978; Lehmann, 2010).

Lehmann (2010) stated that there were issues with the use of Grounded Theory in Information Systems (IS) research. He suggested that although Grounded Theory use has increased in IS research over the last two decades the number of grounded theories created has not. He proposed two reasons for this: either Grounded Theory was not a suitable method for IS research or that the method was not performed correctly by researchers.

On the first point, Lehmann (2010) concluded that although Glaser’s and Strauss and Corbin’s Grounded Theory methods originated from social science research investigating interactions between individual humans in non-business environments, the methods could be legitimately adapted to IS research and the interactions between individuals or groups of people within organisations and using different technologies. Lehmann’s (2010) proposed approach, which he claimed legitimised Grounded Theory in the IS domain, was to break down the analysis to focus upon individual experiences within organisations or groups, amalgamating the results into one Grounded Theory.

Lehmann’s (2010) second point, in which he suggested that Grounded Theory was not applied correctly in IS settings, could have a bearing on the outcome of this study. Lehmann explained that abstracting codes and concepts from data is the inductive part of Grounded Theory analysis. The deductive element of Grounded Theory is used to derive from induced codes and concepts the clues and directions for where to go next in the theoretical sampling stage.
For this Grounded Theory analysis the failing to further explore areas, such as those relating to the *Cultural Issues* category (see Section 5.8.6), could be attributed to the researcher’s lack of experience in the Grounded Theory method. For example, additional focus should have been centred on cultural aspects as these were identified in the first interview. However, this theme was not followed-up in subsequent interviews. On reflection, this was probably caused by the researcher’s perception that technical issues *should* be the main areas of concern. Therefore, it can be seen that the success of Grounded Theory is reliant on the researcher’s experience and skill in being able to ignore pre-conceived ideas and to explore areas which are hinted at within the data with confidence. Furthermore, in this study the researcher was an employee of IBM and was familiar with IBM software development methods and knowledge of components. Thus, removing the effect on the analysis of these pre-conceived influences may not have been possible.

It could be suggested that given Grounded Theory’s roots in sociological research and for the investigation of social processes and underlying social behaviour (Charmaz, 2006) for this study it may have been more relevant to use Grounded Theory for the investigation of *Cultural Issues* rather than for the technical issues. However, this suggests a ‘chicken and egg’ dilemma because it was the use of Grounded Theory in the first place which enabled the area of *Cultural Issues*, as a contributing factor to practitioners’ ability to control costs, to emerge in the first place. Other research methods may not have identified *Cultural Issues* as an area of concern.

However, a closer examination of the technical areas of CBD, such as those relating to architectural design, complexity and change activity, suggests that these tasks also require human interaction and the cooperation of people working within teams from the same organisation and between different organisations. As such, these areas may also be influenced by social processes affecting other areas of life. Therefore, if Grounded Theory is deemed suitable for research in sociology it should also be valid for research in CBD which in essence is a human activity.

During this study data was collected using semi-structured interviews. This data collection method was chosen because it was considered to be flexible enough to allow interviewees the freedom to express their views on the area of study, without being too prescriptive and providing a specific list of questions. Adolph et al. (2008) stated that
the semi structured interview data collection method was employed in most of the Grounded Theory studies they reviewed.

However, in hindsight, it can be seen that a dilemma existed. Glaser (1978) suggested that introducing ‘structure’ to the start of the data collection process could introduce a level of preconception. Thus, Glaser’s (1978) view was to allow interviewees the freedom to talk about whatever they considered important; a level of structure would stifle this. However, on the other hand, when using the Grounded Theory method to investigate a specific area of concern, such as the case in this study, some form of structure may be required in order to encourage interviewees to talk about the area of enquiry. Furthermore, the concept of repeatability within scientific enquiry (Coolican, 2009) supports the use of a structured approach to enable other researchers to repeat the study by adopting the same area of questioning. However, with Glaser’s (1978) unstructured view on data collection repeatability of a study would be difficult.

However, once the focus of the study was identified (the Core Category) Glaser (1978) implied that a degree of structure was required to direct the collection of further data. In the process he termed Theoretical Sampling Glaser (1978, p. 36) stated that once the Core Category had been defined to “use the codes to direct further data collection.” Therefore, with Glaser’s Grounded Theory method it can be seen that if a researcher selects the ‘wrong’ Core Category this may influence the rest of the data collection and analysis.

There may have been a greater potential for a new theory to emerge if data was collected from a wider selection of practitioners involved in different parts of the CBD process. For example, one practitioner’s role, such as IT architect, may not have full visibility of all aspects of the integration or maintenance processes and real-life problems experienced by other practitioners, such as integrators, who were actually performing integration or maintenance tasks. Thus, data could have been collected from a wider selection of practitioners, as well as from component vendors and even the customers, for which the systems were being produced. Glaser (1978) stated that the analysis of one data source should provide a lead on where to go next. However, for this study all of the practitioners who contributed to the interview sessions were identified and booked in advance of the analysis. This was done for pragmatic reasons as the practitioners were very busy and finding free time when they could participate proved
challenging. Thus, adhering to Glaser’s (1978) method may have resulted in other collection avenues being pursued, such as staff members from different parts of the business.

Charmaz (2006) discussed collecting ‘rich data’, whereby varied data sources, such as documentary evidence and observer participation could have contributed to the richness of the data sources. Adolph et al. (2008) were critical of Grounded Theory studies which relied solely on interview data. They suggested that interview data would only provide a superficial and recollective description of the phenomena; social interaction should be studied through first-hand observation. This view was supported by Glaser (1998, p. 109) who stated that “in Grounded Theory there is no such thing as observation without interviews to give them meaning. The reverse is also true”. However, for this research project limiting factors concerned the availability of practitioners at the time of data collection and lack of ‘observer participation’ opportunities. Furthermore, documentary sources of evidence were not made available to the researcher.

The implications arising from two other studies should be explored further. For example, Munro and Stansbury (2009) suggested that where the results of research support someone’s own views then they were more likely to agree that the scientific method is suitable to investigate this area. However, when the results of a study differed from a person’s own beliefs then Munro and Stansbury (2009) found that a person would be more likely to reject science’s ability to explain the area of concern, rather than to change their views in light of the study. Thus, this could have explained why in this study the IBM practitioners interviewed for the Grounded Theory analysis believed the *CBS Functional Density Rule of Thumb* (Abts, 2002; 2004) to be true when the statistical data indicated otherwise. However, further research would be required to confirm this.

The second study which could have a bearing on knowledge elicitation was that of Beattie, Webster and Ross (2009). Their research suggested that the interpretation of what people say in language and the physical gestures they make may not be the same. For example, people being interviewed may say one thing but their gestures indicate the opposite. Therefore, in a Grounded Theory analysis the visual cues from people should also be considered as people may be saying one thing which, for example, supports the
company line, but their body language expresses a different view. However, this could not be applied in cases where interviews were conducted over the telephone.

### 7.7 Concluding Thoughts

In hindsight, the expectation that a hybrid Grounded Theory approach, based on Glaser’s (1978) method, would produce a ‘novel’ theory to explain the results of the statistical analysis was flawed. However, the use of the hybrid Grounded Theory approach to analyse interview data and to make sense of large amounts of data can be seen to be valid as long as the output is not claimed to be a Grounded Theory. These conclusions can now be made for the following reasons:

1) **Preconception.** Glaser (1978) stated that a Grounded Theory study using his method must commence before a literature review in order to prevent the researcher being ‘contaminated’ with preconceived ideas. He also stated that the problem statement should also be allowed to emerge from the study. In this study, both of these factors were contravened; a literature review was conducted at the start of the study and a problem area identified – the factors affecting the cost of developing and maintaining COTS-based systems.

Thus, if this study were to be performed again using the Glaserian Grounded Theory method it would be more appropriate to perform the Grounded Theory analysis before conducting the literature review and identifying the focus area. However, this questions the validity of using Glaser’s Grounded Theory method to study any pre-defined study area as this could be seen as forcing the study focus. Thus, in order to focus upon the cost factors of CBD or to validate or reject the *CBS Functional Density Rule of Thumb* (Abts, 2002; 2004), using Glaser’s (1978) Grounded Theory method, then these areas would have needed to emerge as areas of concern from the practitioners and other sources of evidence (which they may not have done).

This highlights a limitation of Glaser’s method; at what point would the selection of the focus area be classed as pre-conception. Glaser (1978) does not make this clear.
The notion of pre-conception also questions the validity of using Glaser’s Grounded Theory method for a PhD study (or any other study) where the normal requirement is for a literature review to occur early on in the process.

Therefore, for this study using Strauss and Corbin’s Grounded Theory method may have been preferable, compared with a hybrid of Glaser’s method because Strauss and Corbin (1998) prescribed starting the Grounded Theory study with a research question (Van Niekerk & Roode, 2009). However, Glaser criticised this feature of Strauss and Corbin’s method, suggesting that it allowed the main concern of subjects to be missed because of preconception (Van Niekerk & Roode, 2009).

2) Mixing Grounded Theory flavours. A problem with using a hybrid Grounded Theory approach for this study concerned the claim that Grounded Theory methods should not be mixed (Van Niekerk & Roode, 2009). For example, researchers new to the Grounded Theory method may see the Glaserian and Straussian Grounded Theory versions as being similar. However, those who try to use a hybrid of the two methods (not appreciating that the two methods are substantially different), only realise after a significant investment in time that the methods are not reconcilable and that either the one or the other should be followed (Van Niekerk & Roode, 2009). Thus, in hindsight and with reference to Van Niekerk and Roode’s (2009) views, it was a mistake to use a hybrid Grounded Theory approach which incorporated only part of Glaser’s method and to expect a valid novel Grounded Theory to emerge. The reason for this statement is that it is now clear that for this study the main feature of Glaser’s Grounded Theory method, preconception, was violated.

However, the apparent misuse of the Grounded Theory method, with the use of hybrid Grounded Theory approaches, was widespread in other studies conducted within the domain of IS. Matavire and Brown (2008) performed a survey of the claimed use of the Grounded Theory method within thirty IS centric research journals (including prestigious journals such as, MIS Quarterly, Information Systems Journal and Communications of the ACM). They found that from the one hundred and twenty six articles, published between 1985 and 2007, 62% of the articles surveyed claimed Grounded Theory analysis techniques without stipulating the particular method used, whether Glaserian, Straussian or other flavours. A further 13% of the articles surveyed used a mixed method approach. Adolph et al. (2008), who also surveyed many research
articles, reported that many of the studies which claimed to use Grounded Theory did not explicitly describe the method they were using. Thus, without knowledge of the Grounded Theory method used it would not be possible to validate the reliability of the results of a Grounded Theory study.

Matavire and Brown (2008) suggested that these trends occurred because of the difficulty of following Glaserian Grounded Theory and that conducting IS research required a flexibility whereby different methods were perceived as appropriate and that the Grounded Theory analysis was not used to produce a grounded theory.

3) Expecting Grounded Theory to generate a theory in areas where theories already exist. The main purpose of the Grounded Theory method is to generate theories in areas where theories do not exist (Glaser, 1978). Furthermore, the Grounded Theory method is geared towards discovering social theory from empirical data sources in a wide range of contexts and activities (Lehmann, 2010). Thus, for this study, which involved an area where other theories had been developed to explain the domain, it may not have been appropriate to expect a new theory to emerge. For example, Yang et al. (2008) identified thirteen cost drivers associated with the production of glue code used to combine COTS components. These included COTS product maturity, COTS integrator personnel capability and COTS Supplier product support. In another example, Yang et al. (2005) identified three primary sources of effort relating to the production of COTS-based systems. These were: COTS product assessment, tailoring and Glue-code development. Furthermore, in domains where other theories already exist it may be possible to determine a theoretical framework to explain these areas of concern by analysing the existing research conducted within these domains. Thus, the use of a difficult method, such as the Ground Theory method may not be the best method to use in these cases (Adolph et al., 2008).

4) Using the Grounded Theory method to explain technical areas. Grounded Theory was developed to explain social processes (Adolph et al., 2008; Lehmann, 2010). Thus, with reference to this it can be suggested that technical domains, not involving social interaction and processes, would not be suitable research areas for the Grounded Theory method. However, it can be seen that CBD involves both technical aspects, such as the compatibility of one component with another and social processes, relating to the interaction and relationships between individuals and teams of people involved with
building systems from COTS components. Therefore, the Grounded Theory method should be valid for research into the social aspects of CBD.

5) Using the Grounded Theory method to analyse data. Whilst it is now clear that the use of a hybrid Grounded Theory approach was not suitable to generate a ‘Grounded Theory’ for this study it could be seen to be a beneficial method from which to make sense of and to interpret the views of CBD practitioners. Adolph et al. (2008) stated that employing the rigor of the Grounded Theory method was valid to develop rich description of an area of concern and that it was not always necessary to produce a Grounded Theory. One benefit of the Grounded Theory method was that because data collection and analysis occurred at the same time the analysis remained fresh in the researcher’s mind. Another benefit was the rigor of the method. The method forced the close examination of the data in order to identify meaning. Constantly comparing concepts and codes contributed to developing an understanding of the areas of concern. This rigor was not a characteristic of alternative data analysis methods, such as content or discourse analysis.
Chapter 8 Discussion of the Findings from this Study

8.1 Introduction

The aim of this chapter is to discuss the findings from this study, with reference to the research objectives and research questions. The aim, objectives and research questions are documented in Chapter 1.

IBM was used as a case study in COTS component use. The study involved the collection and analysis of two separate sources of data. Details of the data collection and analyses are presented in Chapter 4, Statistical Analysis and Chapter 5, Grounded Theory.

The next section presents a discussion of the findings from the Statistical Analysis and Grounded Theory sections.

8.2 Review of the CBS Functional Density Rule of Thumb and Research Findings

The first point to consider was that the findings from the Grounded Theory analysis were not supported by the results of the Statistical Analysis. For example, the practitioners interviewed for the Grounded Theory analysis agreed with the CBS Functional Density Rule of Thumb (Abts, 2002; 2004).

The practitioners interviewed during the Grounded Theory analysis believed that system development costs would be reduced by building more of a system from COTS components because development effort would be ‘time-shifted’ to the vendor (i.e. when a customer selects a COTS component the vendor has already expended effort to develop the product in the first place).

However, the Grounded Theory analysis suggested that there were other factors which needed to be considered to reduce costs. In essence, the practitioners felt that development and maintenance costs were influenced by reducing the number of components used in a system because this contributed to reducing system complexity as
fewer interfaces were required to be integrated and configured. Other factors which were identified to contribute to reducing costs were:

- selecting loosely coupled components, which facilitated changing components with minimum impact on the system as a whole;
- choosing components which supported the same architectural standard, as these components were more likely to integrate together with minimum effort;
- sourcing components from a minimum number of vendors, which contributed to reducing the complexity of the relationships which needed to be maintained between organisations.

However, the results of the Statistical Analysis did not support the **CBS Functional Density Rule of Thumb** (Abts, 2002; 2004). Abts (2002, p. 5) stated that to “minimise development costs the percentage of system functionality delivered via COTS components should be increased”.

The test of Hypothesis 1 (see Section 4.8) did not find a significant association between **Percentage of COTS components** and **Development effort**, measured on a logarithmic scale - Log(Development effort). A multiple regression analysis did not find **Percentage of COTS components** to be a significant predictor of Log(Development effort). With reference to the **CBS Functional Density Rule of Thumb** (Abts, 2002; 2004) it would have been expected that the association between **Percentage of COTS components** and Log(Development effort) would have been stronger.

Therefore, the results originating from the test of Hypothesis 1 brought into question the relationship between the Statistical Analysis section and the Grounded Theory analysis and which analysis was the most reliable. This was because the practitioners interviewed for the Grounded Theory analysis expressed support for the **CBS Functional Density Rule of Thumb** (Abts, 2002; 2004) but the Statistical Analysis results did not support this.

Firstly, there was concern of the nature of the evidence originally used to develop the **CBS Functional Density Rule of Thumb** (Abts, 2002; 2004). Abts (2002, p. 4) stated that the **CBS Functional Density Rule of Thumb** evolved from “anecdotal evidence”
collected from interviews performed to gather calibration data for the extension of the COCOMO cost estimation model. However, details of the *anecdotal evidence* were not provided by Abts (2002) or whether ‘cost related’ figures were used.

Furthermore, there was no evidence to ascertain whether the *CBS Functional Density Rule of Thumb* (Abts, 2002; 2004) had been subsequently tested with empirical evidence. For example, Clark and Clark (2007) only remarked on the ‘maintenance cost’ aspects of COTS-based systems, rather than the cost to develop such systems. They stated that the ‘idea’ of the *CBS Functional Density Rule of Thumb* was “verified when we kept hearing that the complexity of maintaining COTS-based systems increases as the number of different COTS components increases” (Clark & Clark, 2007, p. 5).

It can be suggested that the Grounded Theory analysis performed in this study, which only analysed interview data, could not move beyond the perceived wisdom of the practitioners interviewed. Thus, because the practitioners believed in the *CBS Functional Density Rule of Thumb* the interview data reflected this belief.

Therefore, for this reason the results from the Statistical Analysis were considered to be more *reliable* than the Grounded Theory analysis; data representing *actual* development effort for a very large sample of system development projects was analysed during the Statistical Analysis, compared with the analysis of *perceived* views of interviewees, which occurred with the Grounded Theory analysis. Thus, the statistical data transcended the subjective views of the selection of practitioners interviewed.

The Statistical Analysis also identified a potential weakness of FPA as an accurate predictor of the effort required to develop and maintain systems built from COTS components. Although the tests of Hypotheses 1, 2 and 3 (see Section 4.8) found Log(System size) to be a significant predictor of Log(Development effort), Log(Maintenance effort) and Log(Total effort), the *practical* associations between the variables were weak. However, the data provided for the Statistical Analysis did not include specific details of each system, such as, how Function Points were calculated for custom code, components and, if relevant, glue code and wrapper code. Furthermore, the data did not include other details, such as, the of number of components used in each system, the types of components used, whether all components in each system supported the same architectural standard, the relative amount of effort spent
configuring, integrating or maintaining specific components and the amount of effort spent developing and maintaining custom code.

The Grounded Theory analysis would have been strengthened with the analysis of additional data sources. Glaser (1978) stated that interview data must be enhanced with the analysis of data collected from other methods, such as documentary data. The justification was that the analysis of one evidence source should be used to support or challenge other sources of evidence (Glaser, 1978). However, the fact that suitable documentary data was not available was an important, but unavoidable limitation of this study.

The Grounded Theory analysis highlighted the complexities and the number of factors which needed to be considered when building systems using COTS components. The analysis also identified other factors, such as Organisational and Cultural Issues, which are not covered by the CBS Functional Density Rule of Thumb (Abts, 2002; 2004). These were important contributions of the Grounded Theory analysis.

The following diagram, Figure 8.1, is a reproduction of the cluster diagram presented in Figure 5.3 and displays an overview of the relationships between the concepts and categories which emerged from the analysis of the eleven interviews.

8.3 Forces Affecting CBD

With reference to Figure 8.1, it was felt that the results of the Grounded Theory analysis could be articulated as a set of ‘forces at work’, providing guidance for areas of further research.

Forces originated from design patterns (Gamma, Helm, Johnson & Vlissides, 2002) and detail the factors which shape a design solution. Appleton (2000, p. 5) stated that a design pattern was “a named nugget of instructive information that captures the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and system of forces”.
Figure 8.11: Cluster diagram providing a visual representation of the relationships between concepts and categories.
Forces “reveal the intricacies of a problem and define the kinds of trade-offs that must be considered in the presence of the tension or dissonance they create” (Appleton, 2000, p. 14).

Forces can also be expressed as *issues, constraints or consequences* and are the results and trade-offs of applying a design solution which, for the domain of software system design, often relate to the cost, effort, time, system flexibility, extensibility or portability of a particular design (Gamma et al., 2002).

The following section provides details of the issues and forces which are deemed to affect CBD. The supporting evidence is drawn from the Literature Review and Grounded Theory sections of this study.

Forces can also be expressed as *issues, constraints or consequences* and are the results and trade-offs of applying a design solution which, for the domain of software system design, often relate to the cost, effort, time, system flexibility, extensibility or portability of a particular design (Gamma et al., 2002).

The following section provides details of the issues and forces which are deemed to affect CBD. The supporting evidence is drawn from the Literature Review and Grounded Theory sections of this study.

### 8.4 Forces Related to Managing Complexity

**MANAGING COMPLEXITY**, which was discussed in Section 5.8.4, was an issue which was deemed to affect both the development and maintenance costs of CBD. Conceptually, the notion of taking a set of components and slotting them together to make a system, then replacing one component with another to modify the system, seemed straight forward. However, with reference to Figure 8.1 and the Grounded Theory analysis it was seen that **MANAGING COMPLEXITY** was a major issue which affected the ability to build and maintain systems. Furthermore, *Complexity* did not just encompass *System Complexity*, but also referenced human relationship complexities (*Relationship Complexity*), which were related to the number of vendors and support organisations required to support system development and maintenance.
activities and the human relationships which needed to be developed and maintained between development, vendor and support organisations.

The Grounded Theory analysis suggested that MANAGING COMPLEXITY involved the management of technical complexities associated with building larger, more complex systems, from increasing numbers of components. This also involved managing human relationship complexities arising from increasing numbers of people, representing different organisations, being involved with developing and maintaining COTS-based systems. Furthermore, the cultural aspects associated with vendor companies and system development and support organisations being spread across different countries or continents can significantly add to the challenges of managing the human complexities (the Cultural Issues will be covered in more detail below). The influence of MANAGING COMPLEXITY on CONTROLLING COST was also discussed in the Grounded Principles section presented in 5.10.1, point a).

The following forces are considered to be associated with MANAGING COMPLEXITY:

**System Complexity** (See Section 5.8.4.1): measured in terms of the number of interrelated factors making up a system is an issue which affects the ability of practitioners to control costs. For example, a consequence of using greater numbers of components to develop systems is the additional effort required to integrate and maintain greater numbers of interfaces.

**Maintenance Complexity** (See Section 5.8.4.2): conceptualised the number of interrelated tasks, relationship and dependencies between tasks, which were performed when maintaining COTS-based systems. Higher degrees of Maintenance Complexity were considered to result in increasing costs as a consequence of the increasing amounts of time and effort required to manage and perform the different tasks.

**Degree of Dependency** (See Section 5.8.4.3): Higher Degrees Of Dependency between components and other system parts can lead to increasing costs because seemingly small changes can require more change activity and additional effort.
Relationship Complexity (See Section 5.8.4.4): The complexity of relationships relates to the number of different people and organisations which have to be dealt with when developing and maintaining systems. For systems comprising of components supplied from more than one vendor, system development teams may be required to spend additional effort in dealing with greater numbers of vendor organisations and both vendor and internal support teams. This contributes to cost. Furthermore, as more parties become involved there are more opportunities for parties to deny responsibility for problems with components, thus challenging system developers’ ability to attribute accountability for the root cause of problems.

8.5 Forces Related to Design Principles

With reference to Figure 8.1 category DESIGN PRINCIPLES emerged from the Grounded Theory analysis (see Section 5.8.3). The basis of this category was that the practitioners aimed to design systems which adhered to design principles; by adhering to design principles they felt that they would realise the perceived benefits of building systems from COTS components, namely, reduced development costs and increased system flexibility (See Grounded Principle b) in Section 5.10.1). Furthermore, DESIGN PRINCIPLES were also considered to facilitate reduced maintenance costs. However, the Grounded Theory analysis identified ‘forces’ which affected practitioner’s ability to implement DESIGN PRINCIPLES. For example, a Design Principle (See Interview A in Appendix B1 and open code Selecting architecturally compatible components PMiA6) which was identified from the interview data was for system developers to select components which supported the same architectural standard. However, there were cases when components which did not support the main system architectural standard were selected in order to provide the required system functionality. Other issues reported were of customers insisting that system developers used specific suites of components, even when they were not deemed to be suitable from a design perspective (see Section 5.8.3.7 Concept Conflicting Design Principles). However, one differentiating issue of CBD, compared with other development approaches, was the influence of the vendor within the design process. Examples were given in the interview data whereby vendors withdrew support for components which were integral to the functionality of systems (See Interview C in Appendix B3 and open codes Commercial viability of components ARiC13 and Attributing importance of business critical
components ARiC14). In such cases, system developers were unable to plan for such occurrences.

The following force is considered to be associated with DESIGN PRINCIPLES:

**Conflicting Design Principles** (See 5.8.3.5): One aim of DESIGN PRINCIPLES is to reduce development and maintenance costs of COTS-based systems. However, issues which can be beyond the control of system developers, such as, the availability of suitable components or specific customer requirements, can result in systems being built which encompass *Conflicting Design Principles*; the result of which can contribute to increasing costs.

### 8.6 Forces Related to Managing Change

From the Grounded Theory analysis MANAGING CHANGE (See Section 5.8.5 and Grounded Principles d) and h) in Section 5.10.1) was considered to be a major challenge for system developers. Although *Change* could constitute revenue generation opportunities when commissioned by a customer, the effort required to address *change* requirements was considered to be a significant contributor to ongoing costs. The practitioners interviewed suggested that most business systems would change considerably during the system lifecycle. However, it can be seen that the concept of *Change* is not specific to CBD as all systems are likely to change at some point. However, the differentiating factor of CBD, compared with other development approaches, was that *Change* could be forced upon system administrators by component vendors, for example, change in the form of upgraded components or vendors withdrawing support for components.

The following forces are judged to be related to MANAGING CHANGE:

**Knock-on-effect** (see Section 5.8.5.4) and **Change Unpredictability** (see Section 5.8.5.5): A major contributing factor to the costs of maintaining COTS-based systems are the forces of *Change* – planning and implementing change requires effort and thus, *Cost*. Therefore, forces which contribute further to the costs of change constitute a *Knock-On-Effect*, whereby one change requires additional change activity to take place.
Change Unpredictability, whereby change activity results in unpredictable problems, such as an application failing to function correctly following a change affecting a system component. The consequence of Change Unpredictability can be the requirement for additional time and effort to be incurred in resolving issues. Other latent consequences of Change Unpredictability can be, for example, the damage to a company’s reputation when a business critical application fails following change activity. Change Unpredictability differs from Knock-on Effect in that the effect of the former is normally unexpected.

Resisting Change (see Section 5.8.5.6): This force relates to the human resistance to the effects of change. Not all change is welcomed by people, especially when it requires a change to established working practices. Custom-built systems can be developed to adhere to organisational and business requirements. This may not be the case with systems built from COTS components whereby the functionality supplied from components may not adhere to a company’s working practices.

Risk (see Section 6.2.1): Managing Risk associated with Managing Change was an additional concept which emerged from the literature review following the Grounded Theory analysis. System change activity can introduce risk to business functions, which can vary in impact depending upon the business impact. Therefore, risk management strategies should be considered which assess and manage the risks of implementing changes to COTS components, especially when a change could result in the failure of business critical applications or processes.

8.7 Forces Related to Organisational and Cultural Issues

It was clear from the Grounded Theory analysis that the issues affecting CBD were not all technical in nature (see Sections 5.8.2 and 5.8.6 and Grounded Principles g) and i) in Section 5.10.1). Human issues were deemed important and added to the complexity of managing the costs of developing and maintaining systems. However, as with MANAGING CHANGE, ORGANISATIONAL ISSUES and CULTURAL ISSUES are not specific to CBD but can be seen to affect all system types. However, one difference which seemed unique to CBD was that system components tended to be
supplied by vendor organisations, which added levels of complexity for system developers and maintainers who had to deal with these organisations.

However, the Grounded Theory analysis failed to fully explore the effect of cultural issues which can affect CBD as a result of system development and vendor organisations implementing strategic off-shoring. Thus, system development teams, support teams and vendor organisations are now more likely to be spread across different countries or continents. Therefore, not only do system developers have to manage the complexities of designing and implementing COTS-based systems they may also have to deal with the challenges of communicating with internal support teams and vendor organisations who speak a range of languages and who also operate within different time zones.

Leon and Davies (2008) stated that a consequence of service delivery organisations employing staff at offshore locations were a number of forces, which inhibited the ability of systems developers and administrators to manage costs. Cultural differences between support organisations, system developers and customers, combined with highly dynamic markets and rapid technological change, were identified as major inhibitors of progress. Black, Gottschalk, Lococo and Moore (2009) explained that IBM has been moving towards a globally integrated service delivery organisation, based on Global Delivery Centres in places such as Bangalore and Shenzen. This move has presented many challenges, some technical, but mostly concerned with people and culture. They included language issues, in both verbal and written forms and the requirement for different work patterns as a result of the difference in time zones. However, IBM is not alone. Other companies have exploited global outsourcing. Thus, for CBD the effect of CULTURAL ISSUES on system developers can be significant when vendors have outsourced component development and support to different global locations.

Woodward et al. (2010) suggested that there was a growing popularity for software development teams to be globally distributed. Cultural issues impact of the successful cooperation of distributed teams. An example of cultural communication differences between US and Japanese team members was that nodding and acknowledgement by Japanese team members did not always indicate agreement. Thus, it can be seen that for CBD additional challenges exist when vendor organisations are also globally distributed.
The following forces are related to **ORGANISATIONAL ISSUES**:

**Support Quality** (see Section 5.8.2.3): The development and maintenance of COTS-based systems can require the support and cooperation of product support teams, which may be from vendor organisations or from within a system developer’s own organisation. Furthermore, Support Quality also relates the degree of knowledge and skill of the different support teams and their ability to provide appropriate product support. However, a consequence of poor Support Quality can be increased effort and cost as a result of problems taking longer to be resolved.

**Denying Responsibility** (see Section 5.8.2.5): When problems occur with components vendors may deny responsibility for the source of problems. As the numbers of vendors supplying products for a system increases the opportunity for vendors to deny responsibility for the source of any problems which arise may increase. In such cases, some vendors may be prepared to attribute the root cause of problem to other vendor’s products. The consequence for system developers can be increased time and effort spent dealing with different vendor organisations to resolve problems.

**Sphere of Influence** (see Section 5.8.2.1): CBD is likely to involve different teams and organisations which participate during system development and maintenance processes. Sphere of Influence is more of an issue within CBD because system practitioners are less likely to be able to exert influence over other organisations, such as vendors, who are not part of the system developer’s own organisation. Thus, a customer (system developer) may not be able to compel vendors to modify their components when the actions of a vendor’s organisation are Beyond the Sphere of Influence of the customer’s organisation. Furthermore, the organisational structure of some large organisations can result in internal teams, such as support teams, being Beyond the Sphere of Influence of other groups within the same organisation.

The following forces are linked to **CULTURAL ISSUES**:

**Lacking Common Understanding** (see Section 5.8.6.3): In environments where system development and maintenance teams, component vendors and support teams comprise of people originating from different countries verbal communication cues or language in the written form may be not be interpreted in the same way by all people.
involved with a project. Furthermore, as a result of the current trend for project teams to be geographically dispersed this force may be more likely to occur. The consequence of such situations is that projects may take longer to complete or product problems may require more time and effort to resolve, thus, contributing to cost. Additional time and effort may be required to ensure that all people concerned with system development, maintenance or product support tasks have a common understanding of the issues involved.

Cultural Misunderstandings (see Section 5.8.6.1): Can lead to the purchase of inappropriate components as a result of misunderstanding the technical description of product functionality written by authors where English is not their primary language.

8.8 Conclusions

The aim of this chapter was to offer an understanding of the results of the Statistical Analysis and Grounded Theory sections of this study.

There was an apparent contradiction between the results of the Statistical Analysis and Grounded Theory. The practitioners interviewed for the Grounded Theory Analysis supported the CBS Functional Density Rule of Thumb (Abts, 2002; 2004). However, the Statistical Analysis did not support this theory. It was suggested that the results of the Statistical Analysis were more reliable in this case because only the interview data collection method was used for the Grounded Theory analysis and the practitioners interviewed were probably tainted by their perceived wisdom.

However, the Grounded Theory analysis was useful in identifying the issues and complexities of CBD. It was clear that building systems from COTS components is a complex task and influenced by many factors.

The Grounded Theory analysis has identified some forces which affect CBD. These are listed below:

- System Complexity;
- Maintenance Complexity;
• Relationship Complexity;
• Conflicting Design Principles;
• Knock-on-effect;
• Resisting Change;
• Risk;
• Support Quality;
• Denying Responsibility;
• Sphere of Influence;
• Lacking Common Understanding;
• Cultural Misunderstandings.

It is clear that more work can be performed within these areas. See the Areas for Further Work section in Chapter 9.
Chapter 9 Conclusions and Recommendations

9.1 Achievements of this Research Programme

The achievements of this research programme were to provide a better understanding of the issues and factors affecting CBD and of the appropriateness of the Grounded Theory Method in the area of Computer Science. These will be discussed further in Section 9.6.

The defining factors differentiating CBD from other development methods are that COTS components tend to be supplied from vendors, which are normally separate organisations from those using the components to build systems. Another factor is that the source code of COTS components is not normally available for modification by those using the components to produce systems. Thus, many of the challenges of the CBD approach relate to:

- The availability of suitable components;
- The integration of components with other components custom and legacy code to form systems;
- The relationships between component vendors and component customers;
- Control by vendors of the evolution of their components and thus, dictating when components should be upgraded or will be withdrawn from the marketplace.

9.2 Resolution of the Research Aim and Objectives and Questions

9.2.1 Research Aim

The aim of this research programme, as proposed in Chapter 1, was to challenge the perceived cost benefits of CBD by investigating the factors which influence the cost of building systems from COTS software components. If CBD is to be the cost-saving alternative to the custom-built approach to building software systems then the factors of this method should be fully understood to enable practitioners to make informed decisions on the most appropriate development method to use.
9.2.2 Research Objectives

The research objectives proposed in Chapter 1 were:

1) To assess if there is evidence to support, refute or extend the *CBS Functional Density Rule of Thumb* (Abts, 2002; 2004). From a design perspective, the *CBS Functional Density Rule of Thumb* recommends that to minimise development costs the greatest proportion of system functionality should be supplied from COTS components. However, to reduce maintenance costs this functionality should be supplied from the least number of components.

The *CBS Functional Density Rule of Thumb* (Abts, 2002, p. 5) is as follows:

“Maximize the amount of functionality in your system provided by COTS components but using as few COTS components as possible”.

2) To understand the appropriateness of the Grounded Theory Method within the Computer Science domain.

3) To understand the factors influencing the costs of COTS-based design.

9.2.3 Research Questions

The research questions proposed were:

1) Is there evidence to support, refute or extend the *CBS Functional Density Rule of Thumb* (Abts, 2002; 2004)?

2) Are COTS-based system lifecycle costs significantly different from custom-built software systems and why?

3) What are the factors influencing the costs of COTS-based systems?

The following sections discuss whether the research aim and objectives have been met and the answers to the research questions.
9.3 Assessment of the Evidence to Support, Refute or Extend the CBS Functional Density Rule of Thumb

There was a contradiction in the evidence obtained with regard to addressing this research objective.

The IBM practitioners interviewed for the Grounded Theory analysis supported the CBS Functional Density Rule of Thumb (Abts, 2002; 2004). The suggestion from the Grounded Theory analysis was that the CBS Functional Density Rule of Thumb underpinned the justification for selecting CBD in the first place. The practitioners interviewed believed that building systems from COTS components would contribute to reducing development costs; thus, the more functionality supplied from COTS components the greater the development cost savings.

However, the Statistical Analysis showed that the CBS Functional Density Rule of Thumb (Abts, 2002; 2004) did not apply in this case study (see Chapter 4). A significant association was not found to exist between Percentage of system functionality provided from COTS components and Development effort (measured on a logarithmic scale).

With reference to the test of the CBS Functional Density Rule of Thumb (Abts, 2002; 2004) it can be suggested that the results from the Grounded Theory analysis are less reliable than the results of the Statistical Analysis. It was felt that the Grounded Theory analysis was unable move beyond the preconceived ideas of the practitioners interviewed (see Chapter 7). It was suggested that the IBM practitioners’ knowledge of CBD theory biased them into believing in Abts’ (2002; 2004) CBS Functional Density Rule of Thumb. Furthermore, Abts (2002) stated that he developed the CBS Functional Density Rule of Thumb with reference to anecdotal evidence. Evidence has not been found to confirm if the CBS Functional Density Rule of Thumb (Abts, 2002; 2004) has been subsequently tested with empirical data.

Thus, with reference to research question 1), Is there evidence to support, refute or extend the CBS Functional Density Rule of Thumb (Abts, 2002; 2004), for this case study, Abts’ CBS Functional Density Rule of Thumb (Abts, 2002; 2004) theory fails in totality.
9.4 Understanding of the Appropriateness of the Grounded Theory Method within the Computer Science Domain

It was clear that the aim of using a hybrid Grounded Theory method based upon Glaser’s Grounded Theory flavour and expecting a novel theory to emerge which explained the results of the Statistical Analysis was flawed (see Chapter 7).

The Grounded Theory method was unable to delve beyond the perceived wisdom of the interviewees. This was especially true for this study, as the sources of data used in the Grounded Theory analysis were confined to only collecting interview data. Glaser (1978) stated that interview data should be supported by observation or other data sources and visa versa.

Grounded Theory methods may not be suitable for generating theories in areas where theories already exist. Thus, if interviewees are not open to exploring new ideas within an area of concern then the Grounded Theory method will be unlikely to find anything new. This was confirmed by Glaser (1978), who explained that the purpose of his Grounded Theory method was to develop theories where none exist.

The hybrid Grounded Theory method based on Glaser’s (1978) Grounded Theory Method, which was used to investigate a predefined area of concern, contravened a main tenet of Glaser’s method – preconception. Glaser (1978) stated that a theory must be allowed to emerge from the analysis. He claimed that commencing an analysis with a predefined focus area would prevent the emergence of new ideas. Furthermore, in the case of this research, a researcher possessing knowledge of the area of concern was unlikely to enter the research area with a ‘blank mind’, further hampering the opportunity for the emergence of new ideas. It could also be suggested that the practitioners interviewed did not have ‘blank-minds’ – they were perhaps closed to new ideas, as imbued by Abts’ CBS Functional Density Rule of Thumb, as a result of perceived wisdom.

Another criticism of the hybrid Grounded Theory method used in this study was that Glaser’s (1978) Grounded Theory method differed from other Grounded Theory methods (Van Niekerk & Roode, 2009) and as such, should not have been mixed with
other Grounded Theory flavours (see Chapter 7). It could be suggested that using a hybrid Grounded Theory approach which only used parts of Glaser’s method reduced the opportunity for a novel theory to emerge from the analysis. For example, conducting the literature review at the start of this study contravened Glaser’s Grounded Theory method. Glaser (1978) claimed that this introduces preconception into a study which clouds the researcher’s mind to new ideas.

However, an advantage of the Grounded Theory analysis carried out in this study concerned the rigor of the Grounded Theory method used. Lehmann (2010) stated that a major benefit of the Grounded Theory method, compared with other qualitative data analysis methods, is the rigor of the method. However, although the Grounded Theory analysis failed to produce a new theory, the analysis identified some ‘Grounded Principles’ (see Section 5.10) and a number of ‘forces’ (see Chapter 8) which contributed to a better understanding of the issues affecting the CBD approach. For example, the regression analysis model used to test Hypothesis 1 (see Chapter 4) only explained 4.2% of the variation of system Development effort, with reference to system size and the Percentage of functionality provided from COTS components. Therefore, the statistical analysis suggested that additional factors must have contributed to the remaining variation of Development effort. Therefore, it could be suggested that the Grounded Theory analysis contributed to identifying these factors. Thus, it was clear from the Grounded Theory analysis that many issues and factors affect CBD, many of them non-technical in nature.

Although the hybrid Grounded Theory method used in this study did not produce a novel Grounded Theory for the reasons given above it can be suggested that the different Grounded Theory methods, the roots of which are in the field of sociology, are appropriate for conducting research into the Computer Science domain when the research area involves human interaction. However, researchers should be clear on which Grounded Theory method will be used from the start of the research process.

9.5 Understanding the factors influencing the costs of COTS-based design

Research questions 2 and 3 (see Section 9.2.3) were proposed to address this research objective. With reference to research question 2, Are COTS-based system lifecycle costs
significantly different from custom-built software systems and why, the Statistical Analysis (see Chapter 4) did not find significant associations between variable *Percentage of system functionality provided from COTS components* and variables *Development effort, Maintenance effort* and *Total effort*. For this case study, there were no associations between *Percentage of system functionality provided from COTS components* and *Development effort, Maintenance effort* and *Total effort* respectively. With reference to the perceived cost benefits of CBD, it would have been expected that systems containing higher *Percentages of system functionality provided from COTS components* would have been associated with lower *Development effort, Maintenance effort* and *Total effort* values.

However, for this case study it was not possible to confirm the *total system lifecycle* costs because only five years worth of maintenance effort figures were available. It was not therefore possible to determine how long the systems were planned to be implemented.

Furthermore, from the Statistical Analysis it was not possible to determine *why* there were no differences between the associations of variable *Percentage of system functionality provided from COTS components* with variables *Development effort, Maintenance effort* and *Total effort*. This may have been addressed if further knowledge of the nature of the systems had been available.

The Grounded Theory analysis provided answers to research question 3, *What are the factors influencing the costs of COTS-based systems?* The Grounded Theory analysis identified a set of ‘Grounded Principles’ and ‘forces’ which contributed to explaining these factors. The ‘Grounded Principles’ identified are as follows (see Section 5.10 for more details):

a) Complexity can affect the total lifecycle costs of COTS-based systems and the ability of practitioners to control costs;

b) The application of Design principles are required to reduce complexity, enable system change and reduce total lifecycle costs of COTS-based systems;
c) The maintenance costs of COTS-based systems may be higher than custom-built systems;

d) Designing systems in a way which allows components to be changed independently of other system parts is required to reduce ongoing costs of COTS-based systems;

e) To reduce total lifecycle costs the functionality of COTS-based systems should be supplied from components which support the same architectural standard;

f) To reduce total lifecycle costs the functionality of COTS-based systems should be supplied from components which are supplied by the minimum number of vendors;

g) The costs to develop and maintain COTS-based systems are not only associated with technical issues. Human problems relating to relationships and cultural influences can affect the total lifecycle costs of COTS-based systems;

h) It may not be possible accurately predict the total life costs of COTS-based systems as a consequence of the uncertainties in the COTS marketplace and probable changes in system requirements;

i) Developing working relationships with vendor support organisations and internal support teams can contribute to reducing costs.

The following ‘forces’, which explained the factors deemed to influence the cost of building systems from COTS components, were identified in Chapter 8. These are:

- System Complexity;
- Maintenance Complexity;
- Relationship Complexity;
- Conflicting Design Principles;
- Knock-on-effect;
- Resisting Change;
- Risk;
• Support Quality;
• Denying Responsibility;
• Sphere of Influence;
• Lacking Common Understanding;
• Cultural Misunderstandings.

9.6 Contribution to Knowledge

The contributions to knowledge arising from this PhD study are as follows:

9.6.1 Provide a better understanding of the issues affecting COTS development

The Statistical Analysis section of this research has contested the \textit{CBS Functional Density Rule of Thumb} (Abts, 2002; 2004). Thus, the perception that system development effort should be reduced by building systems where greater percentages of system functionality are provided from COTS components was not supported.

Furthermore, the Grounded Theory analysis provided a better understanding of the issues affecting CBD, articulated as a set of ‘Grounded Principles’ and ‘forces’. It was clear from this analysis that many of the issues affecting CBD were not technical but related to the complexities of human interactions.

9.6.2 Enable a better understanding of the approach, strengths and weaknesses of the Grounded Theory method

For this study, although expecting the hybrid Grounded Theory method to produce a novel theory which explained the results of the Statistical Analysis was flawed this study has identified some strengths and weaknesses of the Grounded Theory method.

For example, regarding the approach of the Grounded Theory method researchers should be aware of the different Grounded Theory flavours and that the main Grounded Theory methods, Glaser (1978) and Strauss and Corbin (1998), are different in approach.
Furthermore, researchers should be clear from the start of a study if they intend using a Grounded Theory method to generate a ‘Grounded Theory’ or only as a data analysis method.

One strength of the Grounded Theory approach concerns the rigor of the different Grounded Theory methods. For Glaser’s method, this rigor relates to the processes of open coding, conceptualisation, constant comparison, memoing, theoretical sorting and theoretical coding which forces the researcher to closely examine the data and to constantly explore the relationships between codes, concepts and categories.

A weakness of the hybrid Grounded Theory method used in this study was its inability to move beyond the preconceived ideas of the practitioners interviewed. A further limitation of this analysis was that only one data source, interview data, was available for analysis. The analysis would have been strengthened if, other data sources, such as, documentary data had been available.

On reflection, a weakness of Glaser’s (1978) and Strauss and Corbin’s (1998) Grounded Theory methods concerned the difficulty in performing their methods. Glaser (1978; 2001) stressed the importance of ‘conceptualisation’ as part of his method. However, Glaser (1978; 2001) did not provide clear instructions on how to perform conceptualisation. Furthermore, it was not clear from Glaser’s (1978) or Strauss and Corbin’s (1998) Grounded Theory methods on how to actually generate the ‘Grounded Theory’ from the preceding Grounded Theory analysis.

9.6.3 Provide a better understanding about the quality of data required to separate different issues

A limitation of the Statistical Analysis section of this study was that only five variables were available (Development effort, Percentage of COTS components, System size, Maintenance effort, Total effort). The Statistical Analysis could have been improved if more details on the systems had been available and if the researcher was given additional insider knowledge of the data. For example, the following details would have improved the understanding of the data:
• In addition to knowing that the systems were Management Information Systems further details of the functions of each system would have been useful;
• Details of the types, numbers and sizes of components used in each system;
• Details of the vendor of each component;
• The system development method used for each system;
• The programming languages used for each system;
• A breakdown of Development effort in terms of tasks performed – custom code development, glue code development, component tailoring, component configuration, problem resolution, component acquisition and testing, time spent dealing with vendors etc.;
• A similar breakdown of Maintenance effort;
• A breakdown of the Development and Maintenance effort in terms of tasks performed and different roles. For example, the amount of effort performed by project managers, programmers, integrators, testers, support personnel etc.;
• A breakdown of the Function Point count for each component, glue code development and custom code development;
• Details of the relative physical locations of the system development staff, support staff, vendor organisations;
• Details of the whole life costs.

The quality of the Grounded Theory analysis would have been improved if the interview data provided from the IBM practitioners was supported by additional data sources, such as documentary data. This view supported by Glaser (1978) who stated that interview data should be supported by additional data sources and visa versa. Charmaz (2006) suggested that a Grounded Theory study should encompass the richness of different data sources. Therefore, a weakness of this Grounded Theory analysis was that it was based solely on the views of the practitioners who were interviewed. Consequently, relying on only this data source may have made it difficult to separate the practitioners views, based upon what they believed to be true, from what was found in the Statistical Analysis.
9.7 Evaluation of this Research Process

This was a successful research programme because the researcher was provided with a substantial sample of data, from a very large multinational company IT company, IBM. The statistical analysis of this data served to challenge the CBS Functional Density Rule of Thumb (Abts, 2002; 2004). However, the research design would have been strengthened if additional details of the systems were available. However, additional details were not available. This was an unavoidable limitation of the study.

The Grounded Theory analysis was successful in identifying some of the issues affecting CBD. However, the decision to use a hybrid Grounded Theory method based on Glaser’s (1978) Grounded Theory method, with the expectation that a novel theory would emerge to explain the results of the Statistical Analysis was flawed. However, the decision to use a hybrid Grounded Theory method, with the expectation that a new theory would emerge, was the result of the researcher’s inexperience with the complexities of the Grounded Theory methodologies. Upon reflection, it is now clear that any decision to use Glaser’s Grounded Theory method must be made very early on in a study, prior to any review of the literature (see Chapter 7). It is also clear that Glaser’s Grounded Theory method should not be selected to examine a specific research problem - Glaser (1978) stated that the research problem must be allowed to emerge.

However, on reflection the hybrid Grounded Theory method proved successful as a data analysis method, identifying issues affecting CBD. Its success was a result of the rigor of the method employed. The constant comparison process combined with memoing, forced the researcher to think about the relationships within the data. However, as mentioned earlier this analysis could have been strengthened if additional data sources, such as documentary data, were made available.

9.8 Areas for Further Research

The areas identified for further research are as follows:

1) Function Point Analysis has been proposed as a reliable predictor of effort. However, although the Statistical Analysis found statistically significant associations between System size measured in Function Points and Development, Maintenance and Total effort, the practical associations between System size and the different measures of effort
were weak. Therefore, further research should be conducted into the reliability of Function Point Analysis as a predictor of system development and system maintenance effort of systems built from components.

2) Abts (2004) stated that Function Points were calibrated on the amount of time required to deliver code from scratch. Thus further research is required to determine if there is parity between the functionality provided from COTS components and that delivered from custom code, with reference to Function Points measured with the IFPUG method.

3) With reference to the Statistical Analysis further research is required to identify additional predictor variables which would account for the variation of system development effort.

4) The data used in this study were collected from Management Information Systems. Further research should be conducted to test the applicability of the CBS Functional Density Rule of Thumb (Abts, 2002; 2004) with other system types. Furthermore, a test of the validity of the CBS Functional Density Rule of Thumb (Abts, 2002; 2004) should be undertaken within other organisations.

5) With reference to Munro and Stansbury’s (2009) study (see Section 7.6) it would be interesting to further explore how people’s perceptions of the ability for scientific methods to change their views when the results of a study do not confirm their preconceptions.

6) Furthermore, with reference to the study performed by Beattie et al. (2009) (see Section 7.6) it would be interesting to investigate further the relationships between the visual cues expressed by people and details of what people say when being interviewed. This would have a bearing on shaping knowledge elicitation methods, such as the interview data collection method in future studies.

7) With reference to the Grounded Principles identified in Section 5.10, further research should be conducted to evaluate their applicability.
8) With reference to the ‘forces’ identified in Chapter 8 further research is required to investigate the following areas:

a) To investigate the relationship between system and human complexities and how they affect the costs to develop and maintain COTS-based systems;

b) To determine the most suitable design principles for the COTS-based approach. This should also consider factors affecting system complexity and the influence of component vendors on the sustainability of the design.

c) Designing For Change (See Section 5.8.5.2) would be an interesting area to investigate further, in the context of designing flexible systems to accommodate the ‘forces of change’, when future change requirements cannot necessarily be predicted when COTS-based systems are being conceived. It is also clear that this research should take into account Designing for Change in conjunction with the management of business risk.

d) To assess the impact of organisational and cultural issues on building systems from COTS components. This is required in response to the recent trend for system development and support organisations to run their business from different off-shore locations.

9.9 Conclusions

This chapter has outlined the successes of this research programme, how the research objectives have been met, outlined the contributions to knowledge and indicated directions for future work.

The aim of this research programme was to challenge the perceived cost benefits of CBD by investigating the factors which influence the cost of building systems from COTS software components. It can be seen that the aim of this research has been achieved. It was shown that there are numerous issues involved with building systems from COTS components; CBD involves many complexities, not all of them technical. Furthermore, the expectation that the greatest development cost savings will be achieved by building systems from the largest percentage of COTS components, as
expressed in Abts’ (2002; 2004) *CBS Functional Density Rule of Thumb*, was not supported.

However, a series of forces influencing the costs of building COTS-based systems, together with a set of ‘Grounded Principles, which when used in combination can enable software practitioners to make informed decisions about the impact on system development costs of using a component-based development approach.
References


Appendix A:

Additional Information for the Statistical Analysis
Appendix A Statistical Analysis

The purpose of this appendix is to present additional information about the data used in the statistical analysis section of this study.

A1 Raw Data Provided by IBM

<table>
<thead>
<tr>
<th>System development effort measured in man hours (Development effort)</th>
<th>Percentage of system functionality provided from COTS components (Percentage of COTS components)</th>
<th>System size measured in Function Points (System size)</th>
<th>System maintenance effort over 5 year period measured in man hours (Maintenance effort)</th>
<th>Total system life effort (development effort plus maintenance effort) measured in man hours (Total effort)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5279</td>
<td>1</td>
<td>547</td>
<td>3156</td>
<td>8435</td>
</tr>
<tr>
<td>252</td>
<td>1</td>
<td>1027</td>
<td>4474</td>
<td>4726</td>
</tr>
<tr>
<td>2770</td>
<td>1</td>
<td>1627</td>
<td>14182</td>
<td>16952</td>
</tr>
<tr>
<td>341</td>
<td>1</td>
<td>1796</td>
<td>2438</td>
<td>2779</td>
</tr>
<tr>
<td>670</td>
<td>1</td>
<td>8952</td>
<td>14174</td>
<td>14844</td>
</tr>
<tr>
<td>5209</td>
<td>1</td>
<td>12801</td>
<td>51870</td>
<td>57079</td>
</tr>
<tr>
<td>2042</td>
<td>1</td>
<td>16078</td>
<td>2090</td>
<td>4132</td>
</tr>
<tr>
<td>1186</td>
<td>5</td>
<td>150</td>
<td>15683</td>
<td>16869</td>
</tr>
<tr>
<td>1761</td>
<td>5</td>
<td>329</td>
<td>4031</td>
<td>5792</td>
</tr>
<tr>
<td>944</td>
<td>5</td>
<td>760</td>
<td>17134</td>
<td>18078</td>
</tr>
<tr>
<td>788</td>
<td>5</td>
<td>3959</td>
<td>7128</td>
<td>7916</td>
</tr>
<tr>
<td>2448</td>
<td>5</td>
<td>11320</td>
<td>41002</td>
<td>43450</td>
</tr>
<tr>
<td>768</td>
<td>10</td>
<td>122</td>
<td>3932</td>
<td>4700</td>
</tr>
<tr>
<td>670</td>
<td>10</td>
<td>211</td>
<td>5362</td>
<td>6032</td>
</tr>
<tr>
<td>2100</td>
<td>10</td>
<td>317</td>
<td>6752</td>
<td>8852</td>
</tr>
<tr>
<td>1351</td>
<td>10</td>
<td>392</td>
<td>28243</td>
<td>29594</td>
</tr>
<tr>
<td>833</td>
<td>10</td>
<td>566</td>
<td>435</td>
<td>1268</td>
</tr>
<tr>
<td>14381</td>
<td>10</td>
<td>640</td>
<td>2024</td>
<td>16405</td>
</tr>
<tr>
<td>6510</td>
<td>10</td>
<td>652</td>
<td>6068</td>
<td>12578</td>
</tr>
<tr>
<td>8467</td>
<td>10</td>
<td>752</td>
<td>58651</td>
<td>67118</td>
</tr>
<tr>
<td>14659</td>
<td>10</td>
<td>880</td>
<td>29636</td>
<td>44295</td>
</tr>
<tr>
<td>550</td>
<td>10</td>
<td>980</td>
<td>21952</td>
<td>22502</td>
</tr>
<tr>
<td>1352</td>
<td>10</td>
<td>1092</td>
<td>19472</td>
<td>20824</td>
</tr>
<tr>
<td>3093</td>
<td>10</td>
<td>1109</td>
<td>29303</td>
<td>32396</td>
</tr>
<tr>
<td>4162</td>
<td>10</td>
<td>1885</td>
<td>24496</td>
<td>28658</td>
</tr>
<tr>
<td>146</td>
<td>10</td>
<td>2147</td>
<td>1856</td>
<td>2002</td>
</tr>
<tr>
<td>System development effort measured in man hours (Development effort)</td>
<td>Percentage of system functionality provided from COTS components (Percentage of COTS components)</td>
<td>System size measured in Function Points (System size)</td>
<td>System maintenance effort over 5 year period measured in man hours (Maintenance effort)</td>
<td>Total system life effort (development effort plus maintenance effort) measured in man hours (Total effort)</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>987</td>
<td>10</td>
<td>2571</td>
<td>1934</td>
<td>2921</td>
</tr>
<tr>
<td>3210</td>
<td>10</td>
<td>2668</td>
<td>5868</td>
<td>9078</td>
</tr>
<tr>
<td>19295</td>
<td>10</td>
<td>3502</td>
<td>78054</td>
<td>97349</td>
</tr>
<tr>
<td>2670</td>
<td>10</td>
<td>5249</td>
<td>5837</td>
<td>8507</td>
</tr>
<tr>
<td>1810</td>
<td>10</td>
<td>11600</td>
<td>12079</td>
<td>13889</td>
</tr>
<tr>
<td>1500</td>
<td>10</td>
<td>12200</td>
<td>34669</td>
<td>36169</td>
</tr>
<tr>
<td>2900</td>
<td>10</td>
<td>39491</td>
<td>16549</td>
<td>19449</td>
</tr>
<tr>
<td>1212</td>
<td>10</td>
<td>39512</td>
<td>25463</td>
<td>26675</td>
</tr>
<tr>
<td>1860</td>
<td>10</td>
<td>53262</td>
<td>18827</td>
<td>20687</td>
</tr>
<tr>
<td>1207</td>
<td>12</td>
<td>300</td>
<td>19461</td>
<td>20668</td>
</tr>
<tr>
<td>4239</td>
<td>15</td>
<td>13306</td>
<td>3217</td>
<td>7456</td>
</tr>
<tr>
<td>305</td>
<td>20</td>
<td>170</td>
<td>194</td>
<td>499</td>
</tr>
<tr>
<td>5903</td>
<td>20</td>
<td>347</td>
<td>17078</td>
<td>22981</td>
</tr>
<tr>
<td>852</td>
<td>20</td>
<td>612</td>
<td>406</td>
<td>1258</td>
</tr>
<tr>
<td>173</td>
<td>20</td>
<td>878</td>
<td>355</td>
<td>528</td>
</tr>
<tr>
<td>15236</td>
<td>20</td>
<td>1108</td>
<td>7010</td>
<td>22246</td>
</tr>
<tr>
<td>2880</td>
<td>20</td>
<td>1236</td>
<td>40440</td>
<td>43320</td>
</tr>
<tr>
<td>890</td>
<td>20</td>
<td>1550</td>
<td>27578</td>
<td>28468</td>
</tr>
<tr>
<td>16983</td>
<td>20</td>
<td>1986</td>
<td>4232</td>
<td>21215</td>
</tr>
<tr>
<td>761</td>
<td>20</td>
<td>2190</td>
<td>6185</td>
<td>6946</td>
</tr>
<tr>
<td>612</td>
<td>20</td>
<td>2376</td>
<td>7628</td>
<td>8240</td>
</tr>
<tr>
<td>1539</td>
<td>20</td>
<td>2456</td>
<td>12728</td>
<td>14267</td>
</tr>
<tr>
<td>500</td>
<td>20</td>
<td>7028</td>
<td>10158</td>
<td>10658</td>
</tr>
<tr>
<td>940</td>
<td>20</td>
<td>23705</td>
<td>6487</td>
<td>7427</td>
</tr>
<tr>
<td>175</td>
<td>25</td>
<td>153</td>
<td>735</td>
<td>910</td>
</tr>
<tr>
<td>3439</td>
<td>25</td>
<td>509</td>
<td>660</td>
<td>4099</td>
</tr>
<tr>
<td>4397</td>
<td>25</td>
<td>1000</td>
<td>7915</td>
<td>12312</td>
</tr>
<tr>
<td>390</td>
<td>25</td>
<td>2655</td>
<td>7058</td>
<td>7448</td>
</tr>
<tr>
<td>983</td>
<td>25</td>
<td>3321</td>
<td>36085</td>
<td>37068</td>
</tr>
<tr>
<td>1700</td>
<td>30</td>
<td>115</td>
<td>198</td>
<td>1898</td>
</tr>
<tr>
<td>237</td>
<td>30</td>
<td>123</td>
<td>205</td>
<td>442</td>
</tr>
<tr>
<td>1310</td>
<td>30</td>
<td>157</td>
<td>1219</td>
<td>2529</td>
</tr>
<tr>
<td>1619</td>
<td>30</td>
<td>195</td>
<td>2218</td>
<td>3837</td>
</tr>
<tr>
<td>517</td>
<td>30</td>
<td>349</td>
<td>497</td>
<td>1014</td>
</tr>
<tr>
<td>5592</td>
<td>30</td>
<td>354</td>
<td>17738</td>
<td>23330</td>
</tr>
<tr>
<td>34</td>
<td>30</td>
<td>543</td>
<td>1041</td>
<td>1075</td>
</tr>
<tr>
<td>System development effort measured in man hours (Development effort)</td>
<td>Percentage of system functionality provided from COTS components (Percentage of COTS components)</td>
<td>System size measured in Function Points (System size)</td>
<td>System maintenance effort over 5 year period measured in man hours (Maintenance effort)</td>
<td>Total system life effort (development effort plus maintenance effort) measured in man hours (Total effort)</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1370</td>
<td>30</td>
<td>700</td>
<td>1920</td>
<td>3290</td>
</tr>
<tr>
<td>2543</td>
<td>30</td>
<td>729</td>
<td>8135</td>
<td>10678</td>
</tr>
<tr>
<td>8342</td>
<td>30</td>
<td>893</td>
<td>17262</td>
<td>25604</td>
</tr>
<tr>
<td>3900</td>
<td>30</td>
<td>955</td>
<td>3967</td>
<td>7867</td>
</tr>
<tr>
<td>116</td>
<td>30</td>
<td>1347</td>
<td>108</td>
<td>224</td>
</tr>
<tr>
<td>322</td>
<td>30</td>
<td>2176</td>
<td>26</td>
<td>348</td>
</tr>
<tr>
<td>949</td>
<td>30</td>
<td>3003</td>
<td>5541</td>
<td>6490</td>
</tr>
<tr>
<td>480</td>
<td>30</td>
<td>10360</td>
<td>5081</td>
<td>5561</td>
</tr>
<tr>
<td>2550</td>
<td>35</td>
<td>25057</td>
<td>8766</td>
<td>11316</td>
</tr>
<tr>
<td>252</td>
<td>40</td>
<td>181</td>
<td>37</td>
<td>289</td>
</tr>
<tr>
<td>6030</td>
<td>40</td>
<td>568</td>
<td>1639</td>
<td>7669</td>
</tr>
<tr>
<td>2235</td>
<td>45</td>
<td>454</td>
<td>9117</td>
<td>11352</td>
</tr>
<tr>
<td>2476</td>
<td>45</td>
<td>888</td>
<td>3697</td>
<td>6173</td>
</tr>
<tr>
<td>3714</td>
<td>45</td>
<td>15347</td>
<td>52572</td>
<td>56286</td>
</tr>
<tr>
<td>85</td>
<td>50</td>
<td>144</td>
<td>37</td>
<td>122</td>
</tr>
<tr>
<td>4198</td>
<td>50</td>
<td>389</td>
<td>21298</td>
<td>25496</td>
</tr>
<tr>
<td>330</td>
<td>50</td>
<td>665</td>
<td>3954</td>
<td>4284</td>
</tr>
<tr>
<td>2714</td>
<td>50</td>
<td>884</td>
<td>3217</td>
<td>5931</td>
</tr>
<tr>
<td>570</td>
<td>50</td>
<td>999</td>
<td>194</td>
<td>764</td>
</tr>
<tr>
<td>2210</td>
<td>50</td>
<td>1800</td>
<td>10101</td>
<td>12311</td>
</tr>
<tr>
<td>4810</td>
<td>50</td>
<td>1823</td>
<td>1065</td>
<td>5875</td>
</tr>
<tr>
<td>880</td>
<td>50</td>
<td>1843</td>
<td>8382</td>
<td>9262</td>
</tr>
<tr>
<td>15315</td>
<td>50</td>
<td>1969</td>
<td>5973</td>
<td>21288</td>
</tr>
<tr>
<td>2240</td>
<td>50</td>
<td>3817</td>
<td>3163</td>
<td>5403</td>
</tr>
<tr>
<td>25054</td>
<td>50</td>
<td>6802</td>
<td>40057</td>
<td>65111</td>
</tr>
<tr>
<td>1240</td>
<td>50</td>
<td>17893</td>
<td>8401</td>
<td>9641</td>
</tr>
<tr>
<td>44070</td>
<td>60</td>
<td>7533</td>
<td>73463</td>
<td>117533</td>
</tr>
<tr>
<td>2404</td>
<td>65</td>
<td>402</td>
<td>3689</td>
<td>6093</td>
</tr>
<tr>
<td>1762</td>
<td>65</td>
<td>5271</td>
<td>24925</td>
<td>26687</td>
</tr>
<tr>
<td>1249</td>
<td>70</td>
<td>567</td>
<td>14075</td>
<td>15324</td>
</tr>
<tr>
<td>347</td>
<td>70</td>
<td>1394</td>
<td>17282</td>
<td>17629</td>
</tr>
<tr>
<td>6613</td>
<td>70</td>
<td>3656</td>
<td>16949</td>
<td>23562</td>
</tr>
<tr>
<td>354</td>
<td>70</td>
<td>7341</td>
<td>12885</td>
<td>13239</td>
</tr>
<tr>
<td>1968</td>
<td>70</td>
<td>51542</td>
<td>3757</td>
<td>5725</td>
</tr>
<tr>
<td>5529</td>
<td>75</td>
<td>591</td>
<td>15447</td>
<td>20976</td>
</tr>
<tr>
<td>28817</td>
<td>75</td>
<td>2787</td>
<td>12120</td>
<td>40937</td>
</tr>
<tr>
<td>System development effort measured in man hours (Development effort)</td>
<td>Percentage of system functionality provided from COTS components (Percentage of COTS components)</td>
<td>System size measured in Function Points (System size)</td>
<td>System maintenance effort over 5 year period measured in man hours (Maintenance effort)</td>
<td>Total system life effort (development effort plus maintenance effort) measured in man hours (Total effort)</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>778</td>
<td>75</td>
<td>21802</td>
<td>36156</td>
<td>36934</td>
</tr>
<tr>
<td>140</td>
<td>80</td>
<td>267</td>
<td>1493</td>
<td>1633</td>
</tr>
<tr>
<td>6671</td>
<td>80</td>
<td>1690</td>
<td>28682</td>
<td>35353</td>
</tr>
<tr>
<td>142</td>
<td>80</td>
<td>1850</td>
<td>985</td>
<td>1127</td>
</tr>
<tr>
<td>2933</td>
<td>80</td>
<td>4632</td>
<td>9738</td>
<td>12671</td>
</tr>
<tr>
<td>5690</td>
<td>80</td>
<td>5801</td>
<td>5135</td>
<td>10825</td>
</tr>
<tr>
<td>12930</td>
<td>80</td>
<td>9297</td>
<td>17633</td>
<td>30563</td>
</tr>
<tr>
<td>12309</td>
<td>80</td>
<td>12752</td>
<td>20983</td>
<td>33292</td>
</tr>
<tr>
<td>380</td>
<td>85</td>
<td>26</td>
<td>518</td>
<td>898</td>
</tr>
<tr>
<td>107</td>
<td>85</td>
<td>246</td>
<td>66</td>
<td>173</td>
</tr>
<tr>
<td>7184</td>
<td>85</td>
<td>804</td>
<td>5386</td>
<td>12570</td>
</tr>
<tr>
<td>7083</td>
<td>85</td>
<td>883</td>
<td>15850</td>
<td>22933</td>
</tr>
<tr>
<td>4216</td>
<td>85</td>
<td>1780</td>
<td>346</td>
<td>4562</td>
</tr>
<tr>
<td>182</td>
<td>85</td>
<td>4655</td>
<td>5672</td>
<td>5854</td>
</tr>
<tr>
<td>1507</td>
<td>90</td>
<td>137</td>
<td>461</td>
<td>1968</td>
</tr>
<tr>
<td>220</td>
<td>90</td>
<td>181</td>
<td>21</td>
<td>241</td>
</tr>
<tr>
<td>6090</td>
<td>90</td>
<td>196</td>
<td>75849</td>
<td>81939</td>
</tr>
<tr>
<td>2472</td>
<td>90</td>
<td>616</td>
<td>6444</td>
<td>8916</td>
</tr>
<tr>
<td>1239</td>
<td>90</td>
<td>805</td>
<td>2378</td>
<td>3617</td>
</tr>
<tr>
<td>2695</td>
<td>90</td>
<td>1511</td>
<td>20444</td>
<td>23139</td>
</tr>
<tr>
<td>760</td>
<td>90</td>
<td>2766</td>
<td>12681</td>
<td>13441</td>
</tr>
<tr>
<td>750</td>
<td>90</td>
<td>3234</td>
<td>1121</td>
<td>1871</td>
</tr>
<tr>
<td>5542</td>
<td>90</td>
<td>3429</td>
<td>49459</td>
<td>55001</td>
</tr>
<tr>
<td>8653</td>
<td>90</td>
<td>4797</td>
<td>9631</td>
<td>18284</td>
</tr>
<tr>
<td>11719</td>
<td>90</td>
<td>5157</td>
<td>53095</td>
<td>64814</td>
</tr>
<tr>
<td>893</td>
<td>95</td>
<td>104</td>
<td>784</td>
<td>1677</td>
</tr>
<tr>
<td>270</td>
<td>95</td>
<td>316</td>
<td>3365</td>
<td>3635</td>
</tr>
<tr>
<td>480</td>
<td>95</td>
<td>352</td>
<td>5019</td>
<td>5499</td>
</tr>
<tr>
<td>1671</td>
<td>95</td>
<td>893</td>
<td>1649</td>
<td>3320</td>
</tr>
<tr>
<td>3089</td>
<td>95</td>
<td>936</td>
<td>3039</td>
<td>6128</td>
</tr>
<tr>
<td>887</td>
<td>95</td>
<td>955</td>
<td>255</td>
<td>1142</td>
</tr>
<tr>
<td>1333</td>
<td>95</td>
<td>1029</td>
<td>5527</td>
<td>6860</td>
</tr>
<tr>
<td>1614</td>
<td>95</td>
<td>1649</td>
<td>21045</td>
<td>22659</td>
</tr>
<tr>
<td>350</td>
<td>95</td>
<td>2338</td>
<td>4914</td>
<td>5264</td>
</tr>
<tr>
<td>920</td>
<td>95</td>
<td>6222</td>
<td>13841</td>
<td>14761</td>
</tr>
<tr>
<td>564</td>
<td>95</td>
<td>7670</td>
<td>15454</td>
<td>16018</td>
</tr>
<tr>
<td>System development effort measured in man hours (Development effort)</td>
<td>Percentage of system functionality provided from COTS components (Percentage of COTS components)</td>
<td>System size measured in Function Points (System size)</td>
<td>System maintenance effort over 5 year period measured in man hours (Maintenance effort)</td>
<td>Total system life effort (development effort plus maintenance effort) measured in man hours (Total effort)</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1615</td>
<td>95</td>
<td>13945</td>
<td>35394</td>
<td>37009</td>
</tr>
<tr>
<td>6134</td>
<td>100</td>
<td>20</td>
<td>44336</td>
<td>50470</td>
</tr>
<tr>
<td>820</td>
<td>100</td>
<td>50</td>
<td>3953</td>
<td>4773</td>
</tr>
<tr>
<td>843</td>
<td>100</td>
<td>56</td>
<td>305</td>
<td>1148</td>
</tr>
<tr>
<td>134</td>
<td>100</td>
<td>84</td>
<td>458</td>
<td>592</td>
</tr>
<tr>
<td>2340</td>
<td>100</td>
<td>167</td>
<td>10178</td>
<td>12518</td>
</tr>
<tr>
<td>372</td>
<td>100</td>
<td>206</td>
<td>613</td>
<td>985</td>
</tr>
<tr>
<td>382</td>
<td>100</td>
<td>300</td>
<td>50306</td>
<td>50688</td>
</tr>
<tr>
<td>2300</td>
<td>100</td>
<td>370</td>
<td>13143</td>
<td>15443</td>
</tr>
<tr>
<td>1170</td>
<td>100</td>
<td>491</td>
<td>1526</td>
<td>2696</td>
</tr>
<tr>
<td>24214</td>
<td>100</td>
<td>532</td>
<td>35125</td>
<td>59339</td>
</tr>
<tr>
<td>2310</td>
<td>100</td>
<td>696</td>
<td>9238</td>
<td>11548</td>
</tr>
<tr>
<td>320</td>
<td>100</td>
<td>745</td>
<td>61</td>
<td>381</td>
</tr>
<tr>
<td>149</td>
<td>100</td>
<td>1176</td>
<td>3400</td>
<td>3549</td>
</tr>
<tr>
<td>2604</td>
<td>100</td>
<td>1319</td>
<td>16721</td>
<td>19325</td>
</tr>
<tr>
<td>148</td>
<td>100</td>
<td>2441</td>
<td>3838</td>
<td>3986</td>
</tr>
<tr>
<td>661</td>
<td>100</td>
<td>3451</td>
<td>651</td>
<td>1312</td>
</tr>
<tr>
<td>695</td>
<td>100</td>
<td>5015</td>
<td>11871</td>
<td>12566</td>
</tr>
<tr>
<td>9126</td>
<td>100</td>
<td>12904</td>
<td>78121</td>
<td>87247</td>
</tr>
<tr>
<td>2312</td>
<td>100</td>
<td>21000</td>
<td>3643</td>
<td>5955</td>
</tr>
<tr>
<td>275</td>
<td>100</td>
<td>24448</td>
<td>6218</td>
<td>6493</td>
</tr>
<tr>
<td>6335</td>
<td>100</td>
<td>30285</td>
<td>41568</td>
<td>47903</td>
</tr>
<tr>
<td>760</td>
<td>100</td>
<td>34797</td>
<td>44256</td>
<td>45016</td>
</tr>
<tr>
<td>1438</td>
<td>100</td>
<td>36690</td>
<td>11055</td>
<td>12493</td>
</tr>
</tbody>
</table>

Table A1: Table presenting the raw data, sorted on variable Percentage of COTS components.
A2 System Size Categories and their Equivalent Function Point Size

The following system size category table was produced from the analysis of system data and Function Points (Software Measurement Services Ltd, 2005).

<table>
<thead>
<tr>
<th>Relative system size category</th>
<th>FP size (measured using IFPUG standard)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extra-extra-small (XXS)</td>
<td>=&gt; 0 and &lt;10</td>
</tr>
<tr>
<td>Extra-small (XS)</td>
<td>=&gt; 10 and &lt;30</td>
</tr>
<tr>
<td>Small (S)</td>
<td>=&gt; 30 and &lt;100</td>
</tr>
<tr>
<td>Medium1 (M1)</td>
<td>=&gt; 100 and &lt;300</td>
</tr>
<tr>
<td>Medium2 (M2)</td>
<td>=&gt; 300 and &lt;1000</td>
</tr>
<tr>
<td>Large (L)</td>
<td>=&gt; 1,000 and &lt; 3,000</td>
</tr>
<tr>
<td>Extra-large (XL)</td>
<td>=&gt; 3,000 and &lt; 9,000</td>
</tr>
<tr>
<td>Extra-extra-large (XXL)</td>
<td>=&gt; 9,000 and &lt; 18,000</td>
</tr>
<tr>
<td>Extra-extra-extra-large (XXXL)</td>
<td>=&gt; 18,000</td>
</tr>
</tbody>
</table>

Table A2a: System size categories and their equivalent Function Point values (Software Measurement Services Ltd, 2005).

IBM Project Size Categorisation by Function Points:

<table>
<thead>
<tr>
<th>Relative system size category</th>
<th>FP size (measured using IFPUG standard)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very small development</td>
<td>Up to 100</td>
</tr>
<tr>
<td>Small development</td>
<td>Around 500</td>
</tr>
<tr>
<td>Normal project development</td>
<td>Around 1000</td>
</tr>
<tr>
<td>Large</td>
<td>Around 5000</td>
</tr>
<tr>
<td>Very Large</td>
<td>10000 and above</td>
</tr>
</tbody>
</table>

Table A2b: IBM System size categories and their equivalent FP size (Peter Thomas, IFPUG Certified Function Point Specialist, personal communication, April 01, 2008).
A3 Bar Diagrams Showing Development, Maintenance and Total Effort by Percentage of COTS Components

Figure A3a: Bar diagram showing the values of Development effort for the systems sorted (from left to right) by Percentage of COTS components.
Figure A3b: Bar diagram showing the values of Maintenance effort for the systems sorted (from left to right) by Percentage of COTS components.
Figure A3c: Stacked bar diagram showing the values of Total effort (Development effort plus Maintenance effort) for the systems sorted (from left to right) by Percentage of COTS.
One purpose for presenting the descriptive statistics was to determine the data’s suitability for parametric tests. The assumptions are that data are normally distributed, variances should be the same throughout the data, should be measured at least at the interval scale and that the data from different systems is independent. For the purpose of this study the parametric tests which were considered for use were Pearson’s test of correlation and regression analysis. The problem with using a parametric test with data which does not meet the required assumptions is that the results can be inaccurate (Field, 2009).

The assumptions of interval and independent data can only be tested by common sense. However, normal distribution can be assessed objectively with a test (Field, 2009), which is presented below.

<table>
<thead>
<tr>
<th>Variable</th>
<th>System development effort measured in man hours (Development effort)</th>
<th>Percentage of system functionality provided from COTS components (Percentage of COTS components)</th>
<th>System size measured in Function Points (System size)</th>
<th>System maintenance effort over 5 year period measured in man hours (Maintenance effort)</th>
<th>Total system life effort (development effort plus maintenance effort) measured in man hours (Total effort)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of records</td>
<td>158</td>
<td>158</td>
<td>158</td>
<td>158</td>
<td>158</td>
</tr>
<tr>
<td>Range</td>
<td>44036</td>
<td>99</td>
<td>53242</td>
<td>78100</td>
<td>117411</td>
</tr>
<tr>
<td>Minimum value</td>
<td>34</td>
<td>1</td>
<td>20</td>
<td>21</td>
<td>122</td>
</tr>
<tr>
<td>Maximum value</td>
<td>44070</td>
<td>100</td>
<td>53262</td>
<td>78121</td>
<td>117533</td>
</tr>
<tr>
<td>Mean</td>
<td>3554.10</td>
<td>52.15</td>
<td>5035.51</td>
<td>13626.26</td>
<td>17180.35</td>
</tr>
<tr>
<td>Standard Error of Mean</td>
<td>462.27</td>
<td>2.87</td>
<td>749.24</td>
<td>1341.96</td>
<td>1605.87</td>
</tr>
<tr>
<td>Median</td>
<td>1523</td>
<td>50</td>
<td>1206</td>
<td>6619.5</td>
<td>10149.5</td>
</tr>
<tr>
<td>Mode</td>
<td>252</td>
<td>10</td>
<td>181</td>
<td>37</td>
<td>122</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>5810.70</td>
<td>36.09</td>
<td>9417.81</td>
<td>16868.23</td>
<td>20185.41</td>
</tr>
<tr>
<td>Variance</td>
<td>3.38</td>
<td>1302.38</td>
<td>8.87</td>
<td>2.85</td>
<td>4.075</td>
</tr>
<tr>
<td>Skewness statistic</td>
<td>3.76</td>
<td>0.07</td>
<td>3.09</td>
<td>1.95</td>
<td>2.18</td>
</tr>
</tbody>
</table>
Table A4a: Descriptive statistics presenting the range, minimum, mean, median, mode, standard deviation, variance, skewness and kurtosis of the raw data for variables: Development effort, Percentage of COTS components, System size, Maintenance effort and Total effort.

Table A4a indicates that the data is not normally distributed because of the large value of the standard deviation for each variable.

The Kolmogorov-Smirnov test of normality (with Lilliefors Significance Correction) was also used. This test compares the data values to a normally distributed set of data with the same mean and standard deviation. The results of this test, using the raw data, are presented in Table A4b.

Table A4b: Results of Kolmogorov-Smirnov test of normality for variables: Development effort, Percentage of COTS components, System size, Maintenance effort and Total effort.

The probability of a normal distribution is assumed if the Kolmogorov-Smirnov test of normality returns a significance level greater than .05 (Coakes et al., 2009). With reference to Table A4b it can be seen that the significance level of this test was less
than .05 (p = .000) for variables Development effort, Percentage of COTS components, System size, Maintenance effort and Total effort. Therefore, normality for the five variables tested could not be assumed.

Although the use of tests of normally can be useful they can have limitations because with large sample sizes significant results can be obtained from small deviations in normality. Therefore, data should also be plotted in order to make informed decisions about the extent of non-normality (Field, 2009). Normal Q-Q plots and histograms of the variables are displayed below.

A4.1 Normal Q-Q Plots and Histograms

Normal Q-Q plot diagrams and histograms of variables: Development effort, Percentage of COTS components, System size, Maintenance effort and Total effort are presented below. For the normal Q-Q plot each observed value was paired with its expected value from a normal distribution. Thus, it would be expected that values of normally distributed variables would fall more or less along a straight line (Coakes et al., 2009).

The purpose of the histogram was to provide a visual representation of the distribution of data points for each variable.
Figure A4a: Normal Q-Q plot of variable Development effort.

With reference to Figure A4a, it can be confirmed that the values for Development effort do not resemble a normal distribution because the data points do not fit along the straight line.

Furthermore, with reference to Figure A4b it can be seen that the majority of systems were associated with a development effort value less than 10,000 man hours, thus indicating a skewed distribution.
Figure A4b: Histogram showing the distribution of variable Development effort.

Figure A4c shows that the data points for variable Percentage of COTS components lie closer to a straight line than the case for the other three variables.
Figure A4c: Normal Q-Q plot of variable Percentage of COTS components.

A histogram showing the distribution of variable Percentage of COTS components is displayed in Figure A4d. This histogram shows that the sample of systems comprised of a Percentage of COTS components ranging from 1 through to 100 and that the distribution is not skewed.
Figure A4d: A histogram showing the distribution of variable *Percentage of COTS components*.

Figure A4e indicates that the values for variable *System size* were not normally distributed because they do not align to a straight line.

Figure A4f shows that the distribution was positively skewed and that most of the systems from the sample were under 5000 Function Points.
Figure A4e: Normal Q-Q plot of variable System size.

Figure A4f: Histogram showing distribution of variable System size.
With reference to Figure A4g it can be seen that the values for variable *Maintenance effort* were not normally distributed because the data points do not align to the straight line.

Figure A4h shows that the distribution of data points for variable *Maintenance effort* were positively skewed.
Figure A4h: Histogram showing distribution of Maintenance effort.

Figure A4i: Normal Q-Q plot of variable Total effort.
Figure A4j: Histogram showing distribution of variable \textit{Total effort}.

It can be confirmed from Figures A4i and A4j that variable \textit{Total effort} was not normally distributed and formed a skewed distribution.
A5 Descriptive Statistics, Normal Q-Q Plots of Variables following Log Transformation (Log base 10)

Figures A5a to A5d provide Normal Q-Q plots for variables Development effort, System size, Maintenance effort and Total effort following the Log transformation (log base 10) of the data. It can be seen that the data points in the Normal Q-Q plots for the transformed data more closely resemble normal distributions. Variable Percentage of COTS components was not transformed.

Figure A5a: Normal Q-Q Plot of Log(Development effort).
Figure A5b: Normal Q-Q Plot of Log(System size).

Figure A5c: Normal Q-Q Plot of Log(Maintenance effort).
Figure A5d: Normal Q-Q Plot of Log(Total effort).
A6 Scatter Diagrams Presenting Plots of the Raw Data

For information, the purpose of this section of the appendix is to present scatter diagrams of plots of the raw data. This is to show how the diagrams differ from the plots of the variables following the logarithmic transformation to base 10.

Figure A6a: Scatter diagram plotting Percentage of COTS components with Development effort.

\[ R^2 \text{ Linear} = 1.223E-4 \]
Figure A6b: Scatter diagram plotting variables Percentage of COTS components and Maintenance effort.
Figure A6c: Scatter diagram plotting System size with Development effort.
Figure A6d: Scatter diagram plotting System size with Maintenance effort.
Figure A6c: Scatter diagram plotting Percentage of COTS components with Total effort.
Figure A6f: System size plotted with Total effort.
A7 PASW Output Tables from Multiple Regression Analysis

Hypotheses 1, 2 and 3 were tested with Multiple Regression Analysis (using the enter method in PASW). PASW presents the results of Multiple Regression Analysis in different tables. These include the following tables: ANOVA, Model Summary and Coefficients table. The purpose of this section is to present the output tables from PASW which were referenced in Chapter 4.

A7.1 Test of Hypothesis 1

**Hypothesis 1**: System size and Percentage of COTS components are significant predictors of Development effort.

**Null Hypothesis 1**: System size and Percentage of COTS components are not significant predictors of Development effort.

Table A7a shows that a significant multiple linear regression model emerged as p < .05 (p = .035).

<table>
<thead>
<tr>
<th>Model</th>
<th>Sum of Squares</th>
<th>Df</th>
<th>Mean Square</th>
<th>F</th>
<th>Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regression</td>
<td>2.303</td>
<td>2</td>
<td>1.152</td>
<td>3.412</td>
<td>.035</td>
</tr>
<tr>
<td>Residual</td>
<td>52.315</td>
<td>155</td>
<td>.338</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>54.618</td>
<td>157</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table A7a: ANOVA table assessing the significance of the multiple linear regression model to determine if Log(System size) and Percentage of COTS components are significant predictors of Log(Development effort).

Table A7b shows that only 4.2% of the variation of Log(Development effort) could be attributed to Log(System size) and Percentage of COTS components.

<table>
<thead>
<tr>
<th>R</th>
<th>R Square</th>
<th>Adjusted R Square</th>
<th>Std Error of the Estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>.205</td>
<td>.042</td>
<td>.030</td>
<td>.58096</td>
</tr>
</tbody>
</table>

Table A7b: Model summary table showing effect of predictor variables Log(System size) and Percentage of COTS components on dependent variable Log(Development effort).

With reference to Table A7c the following fitted regression model emerged. This is calculated from the Unstandardized coefficients, B column:
Log(Development effort) = 2.692 - 6.844 Percentage of COTS components + 0.165 Log(System size).

<table>
<thead>
<tr>
<th>Source</th>
<th>Unstandardised Coefficients</th>
<th>Standardised Coefficients</th>
<th>t</th>
<th>Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Constant)</td>
<td>2.692</td>
<td>.225</td>
<td>11.985</td>
<td>.000</td>
</tr>
<tr>
<td>Percentage of COTS</td>
<td>-6.844</td>
<td>.001</td>
<td>-.042</td>
<td>.596</td>
</tr>
<tr>
<td>components</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Log(System size)</td>
<td>.165</td>
<td>.065</td>
<td>.199</td>
<td>.013</td>
</tr>
</tbody>
</table>

Table A7c: Table of Coefficients for dependent variable Log(Development effort).

A7.2 Test of Hypothesis 2

Hypothesis 2: System size and Percentage of COTS components are significant predictors of Maintenance effort.

Null Hypothesis 2: System size and Percentage of COTS components are not significant predictors of Maintenance effort.

Table A7d shows that a significant multiple linear regression model emerged as p < .05 (p = .000).

<table>
<thead>
<tr>
<th>Model</th>
<th>Sum of Squares</th>
<th>df</th>
<th>Mean Square</th>
<th>F</th>
<th>Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regression</td>
<td>16.715</td>
<td>2</td>
<td>8.357</td>
<td>16.964</td>
<td>.000</td>
</tr>
<tr>
<td>Residual</td>
<td>79.359</td>
<td>155</td>
<td>.493</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>93.074</td>
<td>157</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table A7d: ANOVA table assessing the significance of the multiple linear regression model to determine if Log(System size) and Percentage of COTS components are significant predictors of Log(Maintenance effort).

Table A7e shows that only 18% of the variation of Log(Maintenance effort) could be attributed to Log(System size) and Percentage of COTS components. Thus, 82% of the variation in Log(Maintenance effort) is explained by other factors.
Table A7e: Model summary table showing effect of predictor variables Log(System size) and Percentage of COTS components on dependent variable Log(Maintenance effort).

With reference to Table A7f the following fitted regression model emerged. This is calculated from the Unstandardized coefficients, B column:

\[
\text{Log(Maintenance effort)} = 2.294 - 4.679 \text{ Percentage of COTS components} + 0.456 \text{ Log(System size)}.
\]

Table A7f: Table of Coefficients for dependent variable Log(Maintenance effort).

A7.3 Test of Hypothesis 3.

**Hypothesis 3:** System size and Percentage of COTS components are significant predictors of Total effort.

**Null Hypothesis 3:** System size and Percentage of COTS components are not significant predictors of Total effort.

Table A7g shows that a significant multiple linear regression model emerged as p < .05 (p = .000).

Table A7g: ANOVA table assessing the significance of the multiple linear regression model to determine if Log(System size) and Percentage of COTS components are significant predictors of Log(Total effort).
Table A7h shows that only 17.5% of the variation of Log(Total effort) could be attributed to Log(System size) and *Percentage of COTS components*. Thus, 82.5% of the variation in Log(Total effort) is explained by factors not included in this model.

<table>
<thead>
<tr>
<th>R</th>
<th>R Square</th>
<th>Adjusted R Square</th>
<th>Std Error of the Estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>.418</td>
<td>.175</td>
<td>.164</td>
<td>.57015</td>
</tr>
</tbody>
</table>

*Table A7h: Model summary table showing effect of predictor variables Log(System size) and Percentage of COTS components on dependent variable Log(Total effort).*

With reference to Table A7i the following fitted regression model emerged. This is calculated from the Unstandardized coefficients, B column:

\[
\text{Log(Total effort)} = 2.787 - 4.747 \text{ Percentage of COTS components} + 0.363 \text{ Log(System size)}.\]

<table>
<thead>
<tr>
<th>Source</th>
<th>Unstandardised Coefficients</th>
<th>Standardised Coefficients</th>
<th>t</th>
<th>Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Constant)</td>
<td>2.787</td>
<td>.220</td>
<td>12.644</td>
<td>.000</td>
</tr>
<tr>
<td>Percentage of COTS components</td>
<td>-4.747</td>
<td>-.027</td>
<td>-3.76</td>
<td>.707</td>
</tr>
<tr>
<td>Log(System size)</td>
<td>.363</td>
<td>.415</td>
<td>5.684</td>
<td>.000</td>
</tr>
</tbody>
</table>

*Table A7i: Table of Coefficients for dependent variable Log(Total effort).*
Appendix B: Grounded Theory Data Collection and Analysis
Appendix B1 - Interview with Project Manager (PMiA)

The following appendices show the analysis of interview data. For each interview the following are included: Interview field notes, key point identifiers and coding and initial memos.

Contact summary form – field notes from interview:

<table>
<thead>
<tr>
<th>Contact Summary Form</th>
<th>Interview A (iA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contact name &amp; position:</td>
<td></td>
</tr>
<tr>
<td>Current position: XXXXXXX.</td>
<td></td>
</tr>
<tr>
<td>Contact Type: Interview, Document etc.:</td>
<td>Telephone interview</td>
</tr>
<tr>
<td>Contact Date:</td>
<td></td>
</tr>
<tr>
<td>Visit:</td>
<td></td>
</tr>
<tr>
<td>Phone:</td>
<td></td>
</tr>
<tr>
<td>NOTES</td>
<td></td>
</tr>
<tr>
<td>1. Main issues and themes discussed:</td>
<td>The cost factors affecting systems built from COTS components</td>
</tr>
<tr>
<td>2. Summary of information gathered (of failed to collect) on each target question:</td>
<td>Beneficial, if possible to build a COTS components with the least number of components so long as they are of the same architectural basis</td>
</tr>
<tr>
<td>3. Other interesting points:</td>
<td></td>
</tr>
<tr>
<td>4. What further information required?</td>
<td></td>
</tr>
</tbody>
</table>

Telephone Interview with James Kile, Project Manager

Question: What are the factors which affect the Total Cost of Ownership of a COTS-based system? How does the number of components used to build a CBS affect the TCO?

Bluepages components used in this application – he considered this to have saved 600 man hours in initial development time.

xxxxxx agreed with the CBS Functional density Rule of Thumb in principle – in his experience the maintenance of multiple interfaces takes taken much more effort and planning. For example, using 50% of the functionality of a large component, which solves 90% of all business problems, is far better than utilising 100% of the functionality of numerous smaller components.

However, integrating more COTS components which have the same architectural basis is better than attempting to integrate fewer components which are architecturally different. Architecturally disparate components require more integration effort. Furthermore, if they are supplied by different vendors there may also be different support agreements and upgrade roadmap policies.

The assessment of COTS components has been a major cost factor. Although the functionality of components is published it can be very time consuming to determine if are actually suitable.
In some instances the terminology used to describe their functionality can be confusing (he gives the example of the differences between US and UK English).

The xxxx system has 4 main COTS components Java Mail, Lotus Domino, IE and DB/2. The remainder of the application was written in C++.

The function point value for this application was higher than some other applications due to the nature of the system – the system enables end-users to generate financial information reports. There are numerous possible report types, ranging from gross product summaries to specific financial deltas. Each report selection creates a db/2 query.

The maintenance hours have been less for this system than for other system types because of the way it has been designed – for example, additional report structures are added to the DB/2 query generation module.

Therefore, the actual application did not require to be changed much.

There is also a difference in understanding of what constitutes system maintenance. Some developers classify/incorporate system enhancement activity as system maintenance. This was not the case with XXXXX project.

One main cost factor affecting COTS systems are the dependencies which were not immediately apparent. For example, upgrading the underlying OS can cause the COTS products not to work correctly.

Prior to 1999 the XXXX application was written in REX for OS/2.

---

**Open coding:**

(Project Manager) interview open coding:

<table>
<thead>
<tr>
<th>CodeNo.</th>
<th>Key Text</th>
<th>Early Open Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMiA1</td>
<td>[COTS components] used in this application</td>
<td>System developers using COTS components</td>
</tr>
<tr>
<td>PMiA2</td>
<td>considered this to have saved 600 man hours in initial development time</td>
<td>Saving development time</td>
</tr>
<tr>
<td>PMiA3</td>
<td>the maintenance of multiple interfaces takes much more effort and planning</td>
<td>Maintaining multiple interfaces Increasing system maintenance effort</td>
</tr>
<tr>
<td>PMiA4</td>
<td>using 50% of the functionality of a large component, which solves 90% of all business problems</td>
<td>Preferring large components Solving business problems</td>
</tr>
<tr>
<td>PMiA5</td>
<td>is far better than utilising 100% of the functionality of numerous smaller components</td>
<td>Architects recommending using fewer components</td>
</tr>
<tr>
<td>PMiA6</td>
<td>integrating more COTS components which have the same architectural basis</td>
<td>Selecting architecturally compatible components</td>
</tr>
<tr>
<td>PMiA7</td>
<td>is better than attempting to integrate fewer components which are architecturally different.</td>
<td>Avoiding architectural incompatibility</td>
</tr>
<tr>
<td>PMiA8</td>
<td>Architecturally disparate components require more integration effort.</td>
<td>Selecting architecturally incompatible components Integration effort</td>
</tr>
<tr>
<td>PMiA9</td>
<td>if they are supplied by different vendors.</td>
<td>COTS supplier issues</td>
</tr>
<tr>
<td>PMiA10</td>
<td>there may also be different support agreements and upgrade roadmap policies</td>
<td>Multiple vendors Conflicting maintenance schedules</td>
</tr>
<tr>
<td>PMiA11</td>
<td>The assessment of COTS components has</td>
<td>Assessing component</td>
</tr>
<tr>
<td>CodeNo.</td>
<td>Key Text</td>
<td>Early Open Codes</td>
</tr>
<tr>
<td>---------</td>
<td>---------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>been a major cost factor</td>
<td>suitability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Increasing system development costs</td>
</tr>
<tr>
<td>PMiA12</td>
<td>Although the functionality of components are published it can be very</td>
<td>Establishing component suitability</td>
</tr>
<tr>
<td></td>
<td>time consuming to determine if they are actually suitable.</td>
<td>Requiring effort</td>
</tr>
<tr>
<td>PMiA13</td>
<td>In some instances the terminology used to describe their functionality</td>
<td>Difficulty understanding terminology</td>
</tr>
<tr>
<td></td>
<td>can be confusing</td>
<td>Interpreting language</td>
</tr>
<tr>
<td>PMiA14</td>
<td>The maintenance hours have been less for this system than for other</td>
<td>Designing for change</td>
</tr>
<tr>
<td></td>
<td>system types because of the way it was designed</td>
<td></td>
</tr>
<tr>
<td>PMiA15</td>
<td>the actual application did not require to be changed much</td>
<td>Designers minimising system change</td>
</tr>
<tr>
<td>PMiA16</td>
<td>There is also a difference in the understanding of what constitutes</td>
<td>Problems classifying system maintenance</td>
</tr>
<tr>
<td></td>
<td>system maintenance</td>
<td></td>
</tr>
<tr>
<td>PMiA17</td>
<td>Some developers classify/incorporate system enhancement activity as</td>
<td>Lacking common understanding of maintenance</td>
</tr>
<tr>
<td></td>
<td>system maintenance</td>
<td></td>
</tr>
<tr>
<td>PMiA18</td>
<td>dependencies which were not immediately apparent</td>
<td>Appreciating system dependencies</td>
</tr>
<tr>
<td>PMiA19</td>
<td>upgrading the underlying OS can cause the COTS products not to work</td>
<td>System dependencies</td>
</tr>
<tr>
<td></td>
<td>correctly</td>
<td></td>
</tr>
</tbody>
</table>

Initial memos:
Memo 1 (Interview)

---

**Comparing codes to codes to generate concepts**

**Concept BALANCING COST CHALLENGES**

**Comparing Increasing system development costs PMiA11 and Integration effort PMiA8**

From the data, *assessing component suitability* encompassed tasks requiring human effort and skill, incurring financial cost as people are paid for their effort and skill level.

Thus, *assessing component suitability* can be a contributing factor to *Increasing system development costs PMiA11* depending on the amount of effort and skill required to perform these tasks.

*Integration effort PMiA8* is human effort incurred when integrating COTS-components and other system parts. *Integration effort PMiA8* can be a contributing factor to *Increasing system development costs PMiA11* because human effort incurs financial cost.

The link between codes *Increasing system development costs PMiA11* and *Integration effort PMiA8* is developers **balancing cost challenges** associated with *Increasing system development costs PMiA11* and *Integration effort PMiA8* when developing...
COTS-based systems.

From the data *Increasing system development costs PMiA11* related to the assessment of COTS component suitability for use in a system. Cost resulted from the human effort required to perform assessment tasks of the suitability of one or more components and for the assessment of the cumulative effect of integrating the component(s) within the rest of a system. Thus, increasing cost was directly linked to increasing effort.

From the data code *Increasing system development costs PMiA11* focused upon the component assessment tasks occurring whilst a system was being developed. However, it can be seen that component assessment tasks can be required throughout the system lifecycle, such as, during the maintenance phase when components are being replaced or upgraded.

*Integration effort PMiA8*, from the data, related to the human effort required to integrate COTS components when developing a system. The integration of architecturally disparate components required more effort than integrating components with the same architectural basis because [as surmised from the data] fewer changes were required to make the components work together ['changing things’ required effort]. However, as with code *Increasing system development costs PMiA11*, code *Integration effort PMiA8* is not just related to the system development phase because Integration effort can be required when integrating new or modified components during the system maintenance phase.

However, with reference to the data both codes, *Increasing system development costs PMiA11* and *Integration effort PMiA8*, can be seen to be related to tasks occurring during the system development phase [from the data the system development phase was differentiated from the system maintenance phase] and therefore, can be seen to be related to concept *System development cost*. For example, the *Increasing* part of code *Increasing system development costs PMiA11* is a property of concept *System development cost* because it indicates that *System development cost* can rise. Another property of *System development cost* is *Reducing system development costs*. This will be discussed later.

Code *Integration effort PMiA8* contributes to system development costs and thus can be seen to be a property of *System development cost*.

**Concept STATEGY FOR REDUCING COSTS**

**Comparing codes System developers using COTS components PMiA1 and Saving development time PMiA2**

*System developers using COTS components PMiA1* to build software systems is a *Strategy for reducing costs* used by system developers to *save development time*. In the system development domain time is money! The longer a development project lasts the greater the wages and other human costs. Furthermore, financial penalties can be incurred if a system is delivered after the contracted date.

Thus, the original concept identification of *Financial cost saving strategy* (see below) has been revised to the more general concept of  *Strategy for reducing costs* – a Financial cost saving strategy can be seen to be a Strategy for reducing costs.
The motivation for System developers using COTS components PMiA1 to build systems is in Saving development time PMiA2 because system developers are able to save on the time to develop code by using COTS components, which have been already been developed by COTS vendors.

From the data there was the assumption that Saving development time PMiA2 would lead to a proportional financial cost saving due to the reduced time developers had to be paid.

Thus, System developers using COTS components PMiA1, is a Financial cost saving strategy with the aim of Saving development time PMiA2.

From the data System developers using COTS components PMiA1 was the crux of CBD. As such, this code was related to the choice of system developers to reuse COTS components. The focus of this code was related to an accepted benefit [by IT Architects, Project Managers and other IT professionals within the company] of COTS-based development. One of the benefits of CBD was reflected in code Saving development time PMiA2.

Saving development time PMiA2 [from the data] was considered to be a major benefit of CBD as system developer personnel’s time [which had a direct link to programmers and other personnel’s wages. Thus, the longer a project took to complete the higher the labour costs] incurred cost. Therefore, it can be seen that Saving development time PMiA2 contributed to the concept of Reducing development cost because time is related to cost.

Note: From the data ‘cost’ was associated with financial cost. The terms ‘time’, ‘effort’ and ‘cost’ were used interchangeably to refer to ‘cost in money’. Thus, if effort or time were saved there was an implied saving in money.

Concept BALANCING COST CHALLENGES
Comparing codes System developers using COTS components PMiA1 and Maintaining multiple interfaces PMiA3

With reference to the data System developers using COTS components PMiA1 and Maintaining multiple interfaces PMiA3 can be linked to developers balancing cost challenges as a result of the trade-off between the benefits of reusing a number of COTS components, such as reducing development costs and higher maintenance costs resulting from maintaining multiple interfaces.

From the data the link between System developers using COTS components PMiA1 and Maintaining multiple interfaces PMiA3 is related to the trade-off performed by system developers when selecting reusable components. COTS components have one or more interfaces, which are the means by which components communicate with other components or system parts. However, successful communication and integration requires the interfaces to adhere to the same ‘standard’ (this may not be the case). Where interfaces do not follow the same standard they may need to be modified to enable communication and integration. A downside of modifying interfaces is that this requires human effort and thus incurs cost. Furthermore, interface modifications may need to be maintained throughout the system lifecycle.

The following trade-off exists: On one hand COTS-based system developers aim to
build systems using as much of the functionality supplied from COTS components in order to reduce development costs. However, the result of using a greater number of components can be a greater number of interconnecting interfaces, which in turn, may require modification during initial integration followed by additional modification during system maintenance, resulting in increasing costs.

Note: System developers are taken to mean the set of people who manage the development (and maintenance) of COTS-based systems.

**Concept BALANCING COST CHALLENGES**
Comparing Saving development time PMiA2 and Maintaining multiple interfaces PMiA3

*Saving development time PMiA2* and *Maintaining multiple interfaces PMiA3* can be linked to developers balancing cost challenges (See memo on System developers using COTS components PMiA1 and Maintaining multiple interfaces PMiA3) because the benefits gained by using COTS components (Saving development time) need to be balanced with additional ongoing costs related to *Maintaining multiple interfaces* (if many components are integrated in the system). Therefore, it can be seen that there is a balance between system developers selecting components which result in saving development time and selecting components which minimise the requirement of maintaining multiple interfaces. If the balance is wrong the human effort, and thus cost, required to maintain multiple interfaces can outweigh the cost benefit of *Saving development time* PMiA2.

**Comparing Saving development time PMiA2 and Increasing system maintenance effort PMiA3**

From the data codes *Saving development time PMiA2* and *Increasing system maintenance effort PMiA3* were linked to concept Conflicting cost challenges because, on one hand, choosing to use COTS components enables system developers to save development time by offsetting the time requirement to write the code themselves. However, due to the volatility of the COTS marketplace using COTS components can result in increasing system maintenance effort (compared with custom code) because over time system administrators have to manage the effects of changing components.

**Concept: STRATEGY FOR REDUCING COSTS**
Comparing Saving development time PMiA2 and Preferring large components PMiA4

The preference of using large components in COTS based systems, reducing the number of interfaces requiring integrating, is a strategy for saving development time and as such, is a strategy for reducing costs employed by system developers.

From the data code, *Preferring large components PMiA4*, expressed COTS system developers’ preference to use fewer larger components, which satisfied most of the system requirements, rather than selecting greater numbers of smaller components which satisfied all system requirements. The suggested benefit of this approach is reduced system complexity because of a lesser number of interconnected interfaces.

Therefore, the relationship between codes *Saving development time PMiA2* and *Preferring large components PMiA4* is a Development effort reducing strategy motivated by system developers in Reducing system complexity by reducing the
Concept: DESIGN OBJECTIVE
Comparing Saving development time PMiA2 and Solving business problems PMiA4

The code Solving business problems PMiA4 expresses one purpose of commercial software systems. In the commercial environment software systems tend to be created to solve business problems. Saving development time PMiA2, to produce systems which solve business problems, is one goal of COTS-system developers and is thus a Design objective.

Comparing Saving development time PMiA2 and Architects recommending using fewer components PMiA5

From the data, code Architects recommending using fewer components PMiA5, was related to IT architects’ perception that building systems from fewer COTS components will reduce system development and maintenance costs because of the reduced complexity of integrating fewer interfaces (The assumption from the data was that connecting more COTS components results in more interfaces – thus, the result of connecting more components together is increasing complexity of interconnecting interfaces).

Therefore, Architects recommending using fewer components PMiA5 is a CBS design principle enabling system developers to save time developing COTS-based systems as a result of fewer components needing to be integrated.

STRATEGY FOR REDUCING COSTS
Comparing Saving development time PMiA2 and Selecting architecturally compatible components PMiA6

Selecting architecturally compatible components PMiA6 requires less integration time and effort, thus resulting in COTS-based system developers saving development time PMiA2.

As previously mentioned time can be related to cost. Given the assumption that Selecting architecturally compatible components PMiA6 can result in saving development time PMiA2 it can be seen to be a strategy for reducing costs employed by system developers.

The basis of code Selecting architecturally compatible components PMiA6 is related to how well components integrate without the need for interface modification. For example, if two components built around different architectural foundations are integrated the format of the data flow from each component’s interface may not be compatible. Therefore, in order to facilitate the integration of both components the format of the data flow between both components has to be modified to enable the data being sent from one component is recognised by the other component. The
human effort required to perform these tasks can be high (because component integrators are writing code to integrate components which were not originally designed to integrate).

Conversely, if two components, which are built around the same architectural basis, or standard, are integrated the interfaces tend not to require modification because they were built to communicate together.

(Standards are a way of reaching agreement among interacting participants. A standard establishes uniform engineering or technical specifications, criteria, methods, processes or practices (Simanta, Lewis, Morris & Wrage, 2008))

Furthermore, the underlying assumption of code Selecting architecturally compatible components PMiA6 was that the number of interconnected components was not the significant cost increasing factor. It could be seen that integrating a greater number of components with the same architectural basis requires less effort, and thus costs less, than integrating fewer components built from different architectural standards because in the former case the effort required to modify component interfaces was not required.

Therefore, the link between codes Saving development time PMiA2 and Selecting architecturally compatible components PMiA6 is the concept of Reducing development cost because by using components supporting the same architectural standard system developers can reduce development time and, thus, financial cost of developing COTS-based systems.

<table>
<thead>
<tr>
<th>STRATEGY FOR REDUCING COSTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparing Saving development time PMiA2 and Avoiding architectural incompatibility PMiA7</td>
</tr>
</tbody>
</table>

Avoiding architectural incompatibility PMiA7, when choosing components to build COTS-based systems, can contribute to Saving development time PMiA2 because less effort and time (and cost) is required to integrate components which share the same architectural basis. Therefore, Avoiding architectural incompatibility PMiA7 is a Strategy for reducing costs employed by system developers by Saving development time.

The basis of code and Avoiding architectural incompatibility PMiA7 relates to developers avoiding integrating components which are built around different architectural standards because they require more integration effort (interface modification).

The comparison of Saving development time PMiA2 and Avoiding architectural incompatibility PMiA7 is related to the comparison of Saving development time PMiA2 and Selecting architecturally compatible components PMiA6 (see above where this was discussed). Therefore, both codes can be seen to be linked to the concept of Reducing development cost.

| Comparing Saving development time PMiA2 and Selecting architecturally incompatible components PMiA8 |

From the data code Selecting architecturally incompatible components PMiA8 relates to developers selecting components which do not support the same architectural
standard.

As discussed in memo Comparing Saving development time PMiA2 and Selecting architecturally compatible components PMiA6 (see above) the use of architecturally incompatible components has negative connotations because they require more effort (than components built to the same architectural standard) to integrate. There is an assumption from the data that architecturally incompatible components can be integrated eventually (however, there may be examples where incompatible components cannot be integrated or that the effort required to force them to integrate outweighs any benefit of selecting them).

From the data there appears to be an inverse link between codes Saving development time PMiA2 and Selecting architecturally incompatible components PMiA8 because the result of developers Selecting architecturally incompatible components is increasing development time as a result of the additional effort required to integrate them.

**Concept REDUCING INTEGRATION EFFORT**
Comparing Saving development time PMiA2 and Integration effort PMiA8

It can be seen from the data that there is a relationship between codes System developers Saving development time PMiA2 and Integration effort PMiA8 because the amount of integration effort affects development time; more integration effort can result in longer development time. Therefore, less integration effort should result in less system development time.

For the purposes of this study (and from the data) ‘integration effort’ is taken to mean ‘enabling one or more system components (COTS components or other system parts) to communicate together to form a software system’.

From the data the assumption is for commercial systems to be developed in as short a time as possible in order to get them to market as soon as possible (obviously related to the details of the contract). Thus, Saving development time PMiA2 is the goal of commercial system developers in order to achieve this. Thus, with COTS-based system development Saving development time PMiA2 can be achieved by developers reducing integration effort by making design decisions leading to easy integration of components.

**Concept - VENDOR HOMOGENEITY**
Comparing Saving development time PMiA2 and COTS supplier issues PMiA9

From the data, code COTS supplier issues PMiA9 was related to the assumption that COTS components supplied by the same supplier support the same architectural standard and that components supplied by different vendors would not (this may not always be the case).

The assumption is that less integration effort is required for components supplied by the same vendor because they support the same architectural standard and are designed to work together, therefore Saving development time PMiA2 for system developers. Thus, both codes can be linked to the concept of vendor homogeneity, implying a commonality of components produced by the same vendors.

**Concept STRATEGY FOR REDUCING COSTS**
Comparing Saving development time PMiA2 and Multiple vendors PMiA10

---

283
The basis of code *Multiple vendors PMiA10* is that COTS-based systems can be built from components supplied by multiple vendors. This forms the basis of the CBD approach which is considered to be analogous to traditional engineering techniques, such as civil engineering where the construction of a bridge does not require the components (nuts, bolts, metal sections etc.) to be supplied by the same supplier.

One benefit of system developers sourcing components from multiple vendors, rather than from one vendor, can be more choice.

With the underlying concept of CBD in mind (see above) the comparison of codes *Saving development time PMiA2* and *Multiple vendors PMiA10* can be seen to be a **Strategy for reducing costs** because with the potential to source and integrate many components from different vendors system developers can save the time, effort and cost of creating the functionality from scratch.

**Concept BALANCING COST CHALLENGES**

*Comparing Saving development time PMiA2 and Conflicting maintenance schedules PMiA10*

An inverse relationship occurs between *Saving development time PMiA2*, which should result in lowering development costs, and *Conflicting maintenance schedules PMiA10*, which can lead to increasing maintenance effort. Thus, successful management of these issues entails system developers and maintainers **Balancing cost challenges** related to saving development time and increasing maintenance cost resulting from conflicting maintenance schedules.

**Concept BALANCING COST CHALLENGES**

*Developers Comparing Saving development time PMiA2 and Assessing component suitability PMiA11*

The link between *Saving development time PMiA2* and *Assessing component suitability PMiA11* can be seen to be **Balancing cost challenges** because, on one hand, the use of COTS components is perceived by COTS system developers to result in *Saving development time PMiA2*. Conversely, *Assessing component suitability PMiA11* contribute to development costs as a result of the human effort required to perform these tasks.

Therefore, COTS-based system developers are balancing the *perceived* cost saving measures of COTS-based development (i.e. building systems from COTS components!) with factors outweighing the cost savings of using COTS components in the first place (i.e. component assessment tasks).

In the data code, *Assessing component suitability PMiA11* is related to the processes performed by COTS system developers to assess the suitability of prospective components. The process of assessing the suitability of candidate components involves two main tasks; verifying if the component performs the desired functions; and verifying that the component will integrate with other parts of the system. From the data the implied assumption of *Assessing component suitability PMiA11* is of increasing cost because the process of component assessment requires human skill and effort which come at a cost.

**Comparing Saving development time PMiA2 and Increasing system development costs PMiA11**
The link between *Comparing Saving development time PMiA2 and Increasing system development costs PMiA11* was discussed above (see *Comparing Saving development time PMiA2 and Assessing component suitability PMiA11*) and relates to the requirement of COTS system developers to be continually balancing development cost saving measures, such as building systems with COTS components and cost increasing factors, such as the spending time and effort assessing the suitability of COTS components.

### Comparing Saving development time PMiA2 and Establishing component suitability PMiA12 and Requiring effort PMiA12

See above memos on *Comparing Saving development time PMiA2 and Assessing component suitability PMiA11* and *Comparing Saving development time PMiA2 and Increasing system development costs PMiA11*.

### Comparing Saving development time PMiA2 and Difficulty understanding terminology PMiA13

From the data code, *Difficulty understanding terminology PMiA13*, relates to the problems system developers have in assessing the suitability of components when vendors use inconsistent terminology to describe the functionality of their components.

From the data vendors publish details of the functionality of their components. However, due to a variety of reasons, such as their country or origin or other cultural factors, there is an inconsistently in the way different vendors describe the capabilities of their components. This adds difficulty to system developers’ ability to assess component suitability because one vendor may describe component functionality differently from another vendor. Thus, the effect on cost is that system developers have to spend more time establishing how components work. Time relates to cost.

Culture is taken to mean: ‘the collective programming of the mind which distinguishes the members of one group or category of people from those of another’ and can apply to nations, organisations, occupations, and professions (Hofstede, 1994, p4). From the interview data the assumption was that cultural relativity, the culture of the human environment in which an organisation operates (Hofstede, 1994) can influence the level of support customers (in this case, system developers are the customers of component vendors) receive.

Therefore, *Saving development time PMiA2* can be achieved by system developers if they address *Difficulty understanding terminology PMiA13*, when assessing components, by appreciating cultural factors of different vendors.

### Comparing Saving development time PMiA2 and Interpreting language PMiA13

Code *Interpreting language PMiA13* relates to the requirement for system developers to interpret the language used by component vendors when describing the functionality of their components.

This code relates to the problem of system developers misinterpreting details of component functionality due to confusion caused by the ambiguity of vendors’ communication of component functionality using the English language. One source of
the problem is difference in meaning of words American English words compared with UK English. Further confusion can occur if component functionality is communicated in English by vendors who are based in countries where English is not the first language.

From the data the assumption of this code is that the choices of language by vendors when describing component functionality have national cultural influences. Hofstede (1994) stated that culture can apply to nations as well as organisations.

The assumption is that code Interpreting language PMiA13 has negative connotations for system developers because interpreting (or misinterpreting) language adds time and effort when performing component selection.

Therefore, initially it appears that codes Saving development time PMiA2 and Interpreting language PMiA13 are not conceptually linked because they can have opposite influences on cost – Saving development time is associated with reducing costs. Interpreting language (and the problems caused by misinterpreting language) can cause system developers to select components which are inappropriate.

However, if the cultural aspects of language interpretation are addressed by system developers, enabling them to understand the terminology differences as described by different vendors then codes Saving development time PMiA2 and Interpreting language PMiA13 can be seen to be linked to concept Appreciating cultural factors because the time saved by interpreting language correctly would be reflected in saving development costs.

**Concept COST REDUCING STRATEGY**

My original thought was that Designing for change PMiA14, if consciously considered, could be seen to be a COST REDUCING STRATEGY employed by system developers aimed at saving development time. However, changing systems is unlikely to affect the development of systems – it is more likely to affect the post-implementation system life-cycle phases when changes tend to occur to systems. **Therefore, these 2 codes are not linked**

**Comparing Saving development time PMiA2 and Designing for change PMiA14**

Code Designing for change PMiA14, from the data, relates to the assumption that all COTS-based systems will be affected by change over time. From the data there are 2 sources of change: 1) Change related to vendor instigated change, such patching or upgrading components; 2) Business related change, which relates to change in the business focus requiring system adaptation to accommodate (such as the withdrawal or addition of components). Therefore, if the premise that because change affects COTS-based systems they should be designed to be changed.

From the data ‘change’ to systems occurs after system implementation – during the system maintenance phase; for something to be changed it needs to be implemented in the first place.

The second assumption from this code is that ‘change’ adds cost. For example, a change, such as upgrading a component, requires effort and skill which include assessing the features of the new component, assessing dependences of the original
component and effects on the dependencies following its upgrade. Upgrading a component can require the redoing of any integration work originally performed during the implementation of the original component. This requires effort and skill. Following an upgrade, testing the component and addressing any problems with the rest of the system also require effort and skill. Change has a bigger impact with increasing levels of complexity as systems get larger because more interdependencies between components can exist.

The link between codes *Saving development time PMiA2* and *Designing for change PMiA14* appears to be concept *Reducing system lifecycle cost* because Saving development time relates to saving development costs and the effect for designing for change is reduced maintenance costs; both development and maintenance cost contribute to system lifecycle costs.

### How to deal with increasing levels of complexity as projects get large (Ed Fries)

**Elementary structures and economy of scale**

<table>
<thead>
<tr>
<th>Comparing Saving development time PMiA2 and Problems classifying system maintenance PMiA16 and Lacking common understanding of maintenance PMiA17</th>
</tr>
</thead>
</table>

Codes, *Problems classifying system maintenance PMiA16* and *Lacking common understanding of maintenance PMiA17*, relate to the differing perception held by system developers on what tasks constitutes ‘system maintenance’, compared with system development or other system lifecycle stages. For example, some developers classify tasks occurring after system implementation as system maintenance whereas others consider changes system parts, whilst the system is being developed, as system maintenance. The result of this can be a massage of monetary figures.

The link between codes *Saving development time PMiA2*, *Problems classifying system maintenance PMiA16* and *Lacking common understanding of maintenance PMiA17* is concept *Lacking common understanding* if software practitioners claim *Saving development time* by mistakenly classifying system development tasks as maintenance tasks.

### Concept CONFLICTING DESIGN DECISIONS

Comparing Saving development time PMiA2 and Appreciating system dependencies PMiA18 and System dependencies PMiA19

From the data codes *Appreciating system dependencies PMiA18* and *System dependencies PMiA19* explain that dependencies can exist between system parts, such as COTS components, operating systems, applications and legacy system parts, etc. One example of a dependency is that a component version may only work with a specific operating system release level.

Therefore, an understanding of system dependencies is required by system developers and maintainers because making a change to one system part, such as applying a patch to the operating system, can cause other system parts to fail or function incorrectly.

Conflicts can exist because the cost benefits accrued from *saving development time* can be offset by higher maintenance effort if resultant system dependencies were not considered in the original system design.
Therefore, codes *Saving development time PMiA2, Appreciating system dependencies PMiA18* and *System dependencies PMiA19* can be linked to concept **conflicting design decisions** because saving development time can result in increasing system maintenance time due to increased system dependency complexity.

<table>
<thead>
<tr>
<th>Concept</th>
<th><strong>INCREASING SYSTEM COMPLEXITY</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Comparing Maintaining multiple interfaces PMiA3 and Increasing system maintenance effort PMiA3</strong></td>
<td></td>
</tr>
</tbody>
</table>

*Maintaining multiple interfaces PMiA3* by system administrators can require *Increasing system maintenance effort PMiA3* due to **increasing system complexity** resulting from an increasing number of interface interconnections between components.

A result of integrating many components in a CBS can be a greater number of interface connections between components compared with systems constructed from fewer components.

*Maintaining multiple interfaces PMiA3* by system administrators can result in *Increasing system maintenance effort PMiA3* because of **increasing system complexity** resulting from the increasing number of connections between components.

<table>
<thead>
<tr>
<th>Concept</th>
<th><strong>CONFLICTING DESIGN PRINCIPLES</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Comparing Maintaining multiple interfaces PMiA3 and Preferring large components PMiA4</strong></td>
<td></td>
</tr>
</tbody>
</table>

**Conflicting design principles** explains the link between codes *Maintaining multiple interfaces PMiA3* and *Preferring large components PMiA4* because the assumption of the former code is that increasing maintenance cost could occur as a result of the increased complexity of systems constructed from many components. The latter code implies a design principle aimed at reducing system complexity by building systems from fewer components.

<table>
<thead>
<tr>
<th>Comparing Maintaining multiple interfaces PMiA3 and Solving business problems PMiA4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>No obvious link</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Concept</th>
<th><strong>CONFLICTING DESIGN PRINCIPLES</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Comparing Maintaining multiple interfaces PMiA3 and Architects recommending using fewer components PMiA5</strong></td>
<td></td>
</tr>
</tbody>
</table>

There is a conflict between codes *Maintaining multiple interfaces PMiA3* and *Architects recommending using fewer components PMiA5*; the underlying assumption motivating architects to recommending using fewer components in a COTS-based system design, thus reducing system complexity. This conflicts with maintaining multiple interfaces, where ‘multiple interfaces’ equates to the integration of many components.

Thus, using fewer components can be viewed as a good design principle which reduces system complexity. Conversely, *maintaining multiple interfaces* can be considered as the consequence of poor design as it can result in additional maintenance effort. Therefore, for the reasons given above both codes can be linked to the concept of **conflicting design principles**.
Concept BALANCING COST CHALLENGES
Comparing Maintaining multiple interfaces PMiA3 and Selecting architecturally compatible components PMiA6

From the data maintaining multiple interfaces PMiA3 can involve activities requiring ongoing effort, adding cost to system maintainers.

By system developers Selecting architecturally compatible components PMiA6 in the first place the effort to maintain multiple interfaces should be less because each individual component is built to integrate with each other, thus requiring less ongoing effort to maintain multiple interfaces. The dependency for this approach is the existence of suitable architecturally compatible components which support system requirements.

The link between the 2 codes is developers balancing cost challenges of maintaining multiple interfaces, which added cost, with the potential for lowering ongoing cost from maintaining multiple interfaces of systems comprising of architecturally compatible components.

From the data it can be seen that Selecting architecturally compatible components PMiA6 enables offsetting cost challengers of Maintaining multiple interfaces PMiA3 of architecturally disparate components.

Concept BALANCING COST CHALLENGES
Comparing Maintaining multiple interfaces PMiA3 and Avoiding architectural incompatibility PMiA7

The link between Maintaining multiple interfaces PMiA3 and Avoiding architectural incompatibility PMiA7 has been modified to Balancing cost challenges because system developers have to balance the increasing costs associated with maintaining multiple interfaces with the lesser costs associated with avoiding architecturally incompatible components.

The integration of a large number of components in a CBS, which can result in the requirement to maintain multiple interfaces, may be the only option for CBS developers when selecting components to satisfy all system requirements.

From the data code Avoiding architectural incompatibility PMiA7 contributes to reducing maintenance cost incurred from Maintaining multiple interfaces PMiA3.

The cost of maintaining systems built from architecturally compatible COTS-based systems should be less than the costs of maintaining systems built from architecturally disparate components because in the former case component interfaces tend to be designed to integrate with other components of the same architectural standard.

From the data the basis of code Avoiding architectural incompatibility PMiA7 is that the number of separate components used to build a system is not the main maintenance cost factor. The cost to maintain systems comprising of more architecturally compatible components can be less than systems built from less architecturally disparate components because of the amount of human effort required to maintain the incompatible interfaces over time.
Therefore, for the reasons above codes *Maintaining multiple interfaces PMiA3* and *Avoiding architectural incompatibility PMiA7* are linked to the concept of **offsetting maintenance cost**.

### Comparing Maintaining multiple interfaces PMiA3 and Selecting architecturally incompatible components PMiA8

Multiple interfaces, resulting from the integration of many components, can add complexity to system architecture because many factors have to be considered when developing a system. System complexity can also be compounded if the components making up the system do not support the same architectural standards because they were supplied by different vendors.

*Maintaining multiple interfaces PMiA3* can be defined in terms of **degree** because multiple varies in number from two through to infinity. However, the assumed association is that multiple interfaces can increase as a result of more interfaces associated with each component or as a result of greater numbers of single-interface components.

Interfaces tend to require maintaining over time. An assumption is that maintenance complexity can change in relation to the number of connected interfaces. Thus, as the number of connected interfaces increases the greater its effect on maintenance complexity because more factors need to be considered, dependencies assessed etc.

The consequence of designers *Selecting architecturally incompatible components PMiA8* on *Maintaining multiple interfaces PMiA3* is to contribute further to maintenance complexity because there are even more factors related to sustaining system stability.

Therefore, the association between *Maintaining multiple interfaces PMiA3* and *Selecting architecturally incompatible components PMiA8* is to contribute to **System maintenance complexity**.

### Comparing Maintaining multiple interfaces PMiA3 and COTS supplier issues PMiA9

No obvious link

### Comparing Maintaining multiple interfaces PMiA3, Multiple vendors PMiA10 and Conflicting maintenance schedules PMiA10

From the data the assumption of codes *Maintaining multiple interfaces PMiA3, Multiple vendors PMiA10* is that system maintenance costs involving maintaining multiple interfaces of components supplied by multiple vendors will be higher than if the components were supplied by the same vendor because the components may not adhere to the same architectural standard.

Additionally, the assumption behind codes *Maintaining multiple interfaces PMiA3* and *Conflicting maintenance schedules PMiA10* is that components supplied by different vendors will not have the same maintenance schedule, thus adding complexity to system maintenance.

Therefore, *Maintaining multiple interfaces PMiA3* of components supplied by *Multiple vendors PMiA10* which have *Conflicting maintenance schedules PMiA10* contributes to concept **system maintenance complexity**.
Note: From now on will perform constant comparison on codes which yeald different concepts.

**Concept DESIGN DECISION**
Comparing Maintaining multiple interfaces PMiA3 and Designing for change PMiA14

From the data it can be seen that Designing for change PMiA14 can reduce the effort, and thus cost, of Maintaining multiple interfaces PMiA3 if the COTS-based system design reflects the requirement for system parts to be changed over time. Thus, codes Maintaining multiple interfaces PMiA3 and Designing for change PMiA14 can be linked to concept design decision if maintaining multiple interfaces is considered by architects when designing a system to be changed.

**Concept BALANCING COST CHALLENGES**
Comparing Maintaining multiple interfaces PMiA3 and Designers minimising system change PMiA15

From the data Maintaining multiple interfaces PMiA3 is associated with cost – the more interfaces the greater the cost. ‘System change’ incurs cost as a result of the human effort and skill required to effect change. Therefore, minimising change lessens this cost.

Therefore, the link between Maintaining multiple interfaces PMiA3 and Designers minimising system change PMiA15 is system developers Balancing cost challenges from maintaining systems built from many components, which satisfy more of the system requirements, and the cost challenges of designing systems in which the requirement for change has been minimised.

From the data it can be seen that by Designers minimising system change PMiA15 the requirement, and thus cost, of Maintaining multiple interfaces PMiA3 can be reduced. Therefore, codes Designers minimising system change PMiA15 can be seen to be linked to concept offsetting maintenance cost of Maintaining multiple interfaces PMiA3.

**Concept INCREASING MAINTENANCE COMPLEXITY**
Comparing Maintaining multiple interfaces PMiA3 and System dependencies PMiA19

From the data the relationship between Maintaining multiple interfaces PMiA3 and System dependencies PMiA19 is Increasing maintenance complexity because with increasing system dependencies the complexity of maintaining multiple interfaces also increases as there are more links for a maintainer to consider, which can increase the likelihood of a problem occurring.

From the data the relationship between Maintaining multiple interfaces PMiA3 and System dependencies PMiA19 is Maintenance effort because the effort in Maintaining multiple interfaces PMiA3 relates to system dependency complexity. Thus, an increase in system dependency complexity can result in increasing maintenance effort of multiple interfaces because the number of things which could go wrong increases.

**Comparing Increasing system maintenance effort PMiA3, Preferring large components PMiA4 and Architects recommending using fewer components PMiA5**
The relationship between *Increasing system maintenance effort* PMiA3, *Preferring large components* PMiA4 and *Architects recommending using fewer components* PMiA5 is a **cost reducing strategy** — the reason for this is because *Increasing system maintenance effort* PMiA3 relates to the cost of maintaining systems comprising of many small components — from key text PMiA3 more components result in increasing maintenance effort because of the increasing number of connections and increasing system complexity. System developer’s and architects strategy of recommending fewer, larger components, which encompass more of the system’s functionality, can help reduce system maintenance cost because, due to the reduced number of connections, fewer components and associated connecting code require changing during system maintenance.

**Concept STRATEGY FOR REDUCING COSTS**

Comparing *Increasing system maintenance effort* PMiA3, *Selecting architecturally compatible components* PMiA6, *Avoiding architectural incompatibility* PMiA7

There is an inverse relationship between code *Increasing system maintenance effort* PMiA3 for COTS-system administrators and codes *Selecting architecturally compatible components* PMiA6 and *Avoiding architectural incompatibility* PMiA7 for system architects/developers. The reason for this is if COTS-system developers do not select architecturally compatible components or fail to avoid architectural incompatibility when developing COTS-based systems the result can be *Increasing system maintenance effort* PMiA3 for system administrators because the integration work required to make architectural disparate components work together may have to be redone when components are upgraded or patched.

Therefore the link between codes *Selecting architecturally compatible components* PMiA6 and *Avoiding architectural incompatibility* PMiA7 is a **strategy for reducing costs** employed by system developers and maintainers to address the possibility of *Increasing system maintenance effort* PMiA3.

**Concept DESIGN DECISION**

Comparing *Increasing system maintenance effort* PMiA3 and *Designers minimising system change* PMiA15

From the data it can be seen that the concept of ‘change’ is a maintenance cost driver; the necessity to effect change to COTS-based systems requires human effort, incurring cost. Thus, more change can result in increasing system maintenance effort.

Therefore, *Designers minimising system change* PMiA15 is links to the concept of a **Design Decision** aimed at addressing *Increasing system maintenance effort* PMiA3.

**Concept STRATEGY FOR REDUCING COSTS**

Comparing *Preferring large components* PMiA4, *Solving business problems* PMiA4 and *Architects recommending using fewer components* PMiA5

From the data *Preferring large components* PMiA4 and *Architects recommending using fewer components* PMiA5 can be seen to be strategies employed by COTS-based system developers to reduce costs. There is a similarity between *Preferring large components* PMiA4 and *Architects recommending using fewer components* PMiA5, the result of which is *fewer larger components* fewer components (logically,
if the aim is to create the same system functionality from a lesser number of components the expectation would be for some of the components to be larger than if creating the same system functionality from a greater number of components). The conscious selection of fewer, larger components can be viewed as a **cost reducing strategy** employed by system developers in order to reduce system complexity and the amount of effort required to manage fewer numbers of interconnecting interfaces.

It can also be seen that the main purpose of developing commercial systems in the first place is in **Solving business problems PMiA4.** Therefore, cost reducing strategies aimed at reducing the costs of developing of maintaining COTS-based systems can also apply to reducing the costs of **Solving business problems PMiA4.**

From the data **Preferring large components PMiA4** and **Architects recommending using fewer components PMiA5** can be strategies employed by COTS-based system developers to reduce costs; integrating fewer, larger components reduces system complexity, and thus development and maintenance human effort, because fewer interconnecting interfaces having to be managed. Therefore, the link between these two codes is a **strategy for reducing costs** employed by system developers.

### Concept BALANCING DESIGN PRINCIPLES

**Comparing Architects recommending using fewer components PMiA5 and Selecting architecturally compatible components PMiA6**

The link between **Architects recommending using fewer components PMiA5** and **Selecting architecturally compatible components PMiA6** is **Balancing design principles,** because, on one hand, architects recommend using the least number of larger components to reduce the number of connections between components. In contrast, an alternative architectural recommendation states that the number of separate components used in the design doesn’t matter as long as they support the same architectural standard; the assumption being that components supporting the same architectural standard are built to integrate easily with each other. However, components supporting both design principles may not be available.

### Concept BALANCING DESIGN PRINCIPLES

**Comparing Architects recommending using fewer components PMiA5 and COTS supplier issues PMiA9**

From the data some IT architects recommend building COTS-based systems from the least number of components as possible to reduce the number of interconnections between components, thus reducing system complexity.

The potential choice of COTS components can be increased if system developers are able to select them from different vendors. In comparison, the choice of components supplied by one vendor may be limited to one area of speciality.

However, a dilemma facing Architects is that the benefit of recommending using fewer components, supplied by different vendors, can be outweighed by additional effort and costs integrating and maintaining components which do not necessarily support the same architectural standard.

Therefore, the link between codes **Architects recommending using fewer components PMiA5** and **COTS supplier issues PMiA9** can be seen to be **balancing design principles** because by attempting to adhere to one ‘good practice’ design principle
architects may have to balance the use of conflicting design principles resulting from the availability of suitable components.

Comparing Architects recommending using fewer components PMiA5 and Assessing component suitability PMiA11

No apparent link

Concept INCREASING MAINTENANCE COMPLEXITY
Comparing Multiple vendors PMiA10 and Conflicting maintenance schedules PMiA10

A result of selecting components from Multiple vendors PMiA10 can be Conflicting maintenance schedules PMiA10 resulting in increasing maintenance complexity because many different factors may need to be considered when planning and performing system maintenance tasks.

Comparing COTS supplier issues PMiA9, Multiple vendors PMiA10 and Difficulty understanding terminology PMiA13

The link between codes COTS supplier issues PMiA9, Multiple vendors PMiA10 and Difficulty understanding terminology PMiA13 can be cultural misunderstandings because developer’s Difficulty understanding terminology PMiA13 can be caused by the cultural differences of multiple vendors, who, potentially, originate anywhere in the world.

Concept DESIGNING FOR CHANGE
Comparing Designing for change PMiA14 and Designers minimising system change PMiA15

COTS-based systems can be susceptible to change factors beyond the control of system administrators. These relate to change being ‘forced’ onto system administrators by component vendors, ranging from component upgrades and patches through to withdrawing components. Change can affect system stability. For example, component change may cause systems not to function correctly. Change requires testing. Problems need resolving. All of which requires human effort to resolve. Therefore, from the data, Designing for change is a requirement for designers of COTS-based systems, whether it involves designers minimising the requirement for system change.

Comparing Difficulty understanding terminology PMiA13 and Interpreting language PMiA13

System developer’s Difficulty understanding terminology PMiA13 can occur as a result of COTS vendors describing technical details of components using their version of English (English may not be the first language of some COTS vendors). A vendor, from one country, may express their interpretation of component functionality using a set of words which can have different meanings to people from other countries.

Therefore, cultural misunderstandings can result from system developers Interpreting language PMiA13 incorrectly and selecting inappropriate components as a result.
Appendix B2 - Interview with Software Architect (ARiB)

Contact summary form – field notes from interview:

<table>
<thead>
<tr>
<th>Contact Summary Form</th>
<th>Interview B (iB)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Contact name &amp; position:</strong></td>
<td>Derek Arnold – Software Architect</td>
</tr>
<tr>
<td><strong>Contact Type:</strong> Interview, Document etc.:</td>
<td>Face to face interview</td>
</tr>
<tr>
<td><strong>Contact Date:</strong></td>
<td>Visit: IBM site</td>
</tr>
<tr>
<td><strong>Phone:</strong></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES**

1. **Main issues and themes discussed:**
   The cost factors affecting systems built from COTS components

2. **Summary of information gathered (of failed to collect) on each target question:**

3. **Other interesting points:**

4. **What further information required?**

**Question:** What are the factors which affect the Total Cost of Ownership of a COTS-based system?

The worst COTS product interviewee has used recently is PayPlus. This was because of the large number of bugs in the product. With Payplus in this solution only 20% of the functionality is used. Therefore, there is a high overhead of unused functionality. However, PayPlus was chosen because it is certified by the FED (the same applied to Hotscan). The existing implementation of Hotscan contains no underlying database. Hotscan had to be integrated with DB/2. This required extra effort compared to the new Hotscan version (6.9). This version of Hotscan came with an integrated version of Oracle. Installation is far easier because the Oracle component is installed silently without user intervention. However, there is an associated Oracle licence fee, which is ultimately passed on to the customer. The additional licence cost is far preferable to the increased effort of integrating the old Hotscan version with DB/2.

With regard to the CBS Functional Density rule interviewee agreed that a system built with fewer COTS components would likely have a lower TCO than a CBS built with more components.

However, interviewee also said that even if integrating a greater number of components which were supplied by the same vendor the TCO could still be higher. However, this factor is dependent upon the size of the vendor’s organisation. For example, with a large vendor, such as Logica, there is appears no synergy between different internal departments. Therefore, when dealing with one vendor supplying many different COTS products the support service can still be a bad as when dealing with different vendors. However, generally, when using fewer components xxxx would expect the TCO to be
This is because:
1. Less interface development and maintenance effort (ARiB19)
2. Reduced integration work (ARiB20)
3. Less ‘vendor relationship’ problems (headaches). (ARiB21)
4. Reduced exposure to product volatility. (ARiB22)

Another factor related to TCO is ‘supplier commitment’. (ARiB23) This is related to the vendor’s commitment to work with the client to resolve any bugs, help resolve integration issues etc. (ARiB24)

With regard to defining a ‘maintenance equilibrium’ value the optimum number of COTS components depends upon the percentage of functionality used within each component. This is because one could be forced to upgrade functional parts of the components not used. (ARiB25)

On this issue Payplus may be replaced by an alternative product, IGT Plus which supplies only the equivalent to 20% of PayPlus’s functionality being used. (ARiB26)

Interviewee defined the system size criteria being used (based upon functional return):
- Small = 1 to 50 Man days (1MD = 7.4 hours)
- Med = 50 to 250 MD
- Large = above 250 MD

Open coding:

(IT Architect) interview open coding:

<table>
<thead>
<tr>
<th>CodeNo</th>
<th>Key Text</th>
<th>Early Open Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARiB1</td>
<td>The worst COTS product xxxxx has used recently is PayPlus</td>
<td>Personal experience</td>
</tr>
<tr>
<td>ARiB2</td>
<td>This was because of the large number of bugs in the product.</td>
<td>Poor component quality</td>
</tr>
<tr>
<td>ARiB3</td>
<td>With Payplus in this solution only 20% of the functionality is used.</td>
<td>Functional redundancy</td>
</tr>
<tr>
<td>ARiB4</td>
<td>PayPlus was chosen because it was requested by the customer and certified</td>
<td>Customer requesting component</td>
</tr>
<tr>
<td></td>
<td>by the FED.</td>
<td>Externally certified</td>
</tr>
<tr>
<td>ARiB5</td>
<td>the same applied to Hotscan</td>
<td>Customer selecting component</td>
</tr>
<tr>
<td>ARiB6</td>
<td>existing implementation of Hotscan contains no underlying database.</td>
<td>Integrating components</td>
</tr>
<tr>
<td></td>
<td>Hotscan had to be integrated with DB/2</td>
<td></td>
</tr>
<tr>
<td>ARiB7</td>
<td>This required extra effort compared</td>
<td>Increasing component</td>
</tr>
<tr>
<td></td>
<td>integration effort</td>
<td></td>
</tr>
<tr>
<td>ARiB8</td>
<td>to the new Hotscan version (6.9)</td>
<td>Vendor supplying upgraded</td>
</tr>
<tr>
<td></td>
<td>components</td>
<td></td>
</tr>
<tr>
<td>ARiB9</td>
<td>This version of Hotscan came with an integrated version of Oracle</td>
<td>Selecting Pre-integrated</td>
</tr>
<tr>
<td></td>
<td></td>
<td>components</td>
</tr>
<tr>
<td>ARiB10</td>
<td>Installation is far easier because the Oracle component is installed</td>
<td>Removing human intervention</td>
</tr>
<tr>
<td></td>
<td>silently without user intervention</td>
<td></td>
</tr>
<tr>
<td>ARiB11</td>
<td>There is an associated Oracle licence fee, which is ultimately passed on</td>
<td>Component licensing fees</td>
</tr>
<tr>
<td></td>
<td>to the</td>
<td></td>
</tr>
<tr>
<td>CodeNo.</td>
<td>Key Text</td>
<td>Early Open Codes</td>
</tr>
<tr>
<td>---------</td>
<td>----------</td>
<td>------------------</td>
</tr>
<tr>
<td>ARiB12</td>
<td>The additional licence cost is far preferable to the increased effort of integrating the old Hotscan version with DB/2.</td>
<td>Increasing component licensing costs Balancing costs</td>
</tr>
<tr>
<td>ARiB13</td>
<td>a system built with fewer COTS components would likely have a lower TCO than a CBS built with more components</td>
<td>Incorporating fewer components Lowering total system costs</td>
</tr>
<tr>
<td>ARiB14</td>
<td>even if integrating a greater number of components which were supplied by the same vendor the TCO could still be higher.</td>
<td>Homogeneity of vendors Increasing total system costs</td>
</tr>
<tr>
<td>ARiB15</td>
<td>this factor is dependent upon the size of the vendor’s organisation</td>
<td>Vendor’s organisational complexity</td>
</tr>
<tr>
<td>ARiB16</td>
<td>with a large vendor, such as xxxxxx, there appeared no synergy between different internal departments</td>
<td>Poor vendor support</td>
</tr>
<tr>
<td>ARiB17</td>
<td>when dealing with one vendor supplying many different COTS products the support service can still be a bad as when dealing with different vendors</td>
<td>Poor vendor support</td>
</tr>
<tr>
<td>ARiB18</td>
<td>generally, when using fewer components Xxxxx would expect the TCO to be lower</td>
<td>Using fewer components Lowering total system costs</td>
</tr>
<tr>
<td>ARiB19</td>
<td>Less interface development and maintenance effort</td>
<td>Reducing component interface development effort Reducing system maintenance effort</td>
</tr>
<tr>
<td>ARiB20</td>
<td>Reduced integration work</td>
<td>Reducing component integration effort</td>
</tr>
<tr>
<td>ARiB21</td>
<td>Less ‘vendor relationship’ problems (headaches).</td>
<td>Reducing vendor involvement</td>
</tr>
<tr>
<td>ARiB22</td>
<td>Reduced exposure to product volatility</td>
<td>Reducing effects of changing components</td>
</tr>
<tr>
<td>ARiB23</td>
<td>Another factor related to TCO is ‘supplier commitment’</td>
<td>Vendor commitment</td>
</tr>
<tr>
<td>ARiB24</td>
<td>This is related to the vendor’s commitment to work with the client to resolve any bugs, help resolve integration issues etc.</td>
<td>Vendor support quality</td>
</tr>
</tbody>
</table>

Initial memos: Memo (Interview)

**Comparing codes to codes to generate concepts**

Component **Conflicting design decisions**
Comparing Poor component quality ARiB2 and Customer requesting component ARiB4

The concept of **conflicting design decisions** links codes Poor component quality ARiB2 and Customer requesting component ARiB4 because an architect would be unlikely to select an unreliable component if it was not specifically requested by the
customer.

This was an example of an architect implementing a conflicting design decision because, on the one hand, he would not normally recommend this component because of its poor quality. However, he was obliged to use the component because of customer coercion; the customer had requested the component. Therefore, the architect was reconciling the use of an unreliable component with customer requirements.

Vendor organisational complexity because of many different departments who do not readily communicate together etc.

### Concept Organisational concerns
Comparing Poor component quality ARiB2 and Vendor’s organisational complexity ARiB15

Poor component quality and vendor’s organisational complexity can be linked to the concept of organisational concerns by system developers because the complexity of a vendors’ organisation can affect the quality and support provided for their components. From the interview data it was seen that poor quality components emanated from organisationally large vendors.

### Concept Organisational concerns
Comparing Poor component quality ARiB2 and Poor vendor support ARiB16, ARiB17

Poor component quality occurs as a result of poor vendor support arising from vendor organisations where there appears little synergy between internal departments and lack of communication and cooperation between departments to facilitate the resolution of bugs and other issues. This problem can be made worse when components span several vendor departments. Therefore, after experiencing a history of poor service and poor component quality architects express organisational concerns with the vendors business.

### Concept Conflicting design principles
Comparing Poor component quality ARiB2 with Homogeneity of vendors ARiB14 and Increasing total system costs ARiB14

Conflicting design principles can result from the effect of Homogeneity of vendors on the total cost of ownership of COTS-based systems because some commentators see incorporating components supplied by the same vendor as a positive design principle aimed at reducing TCO whilst others suggest that as the complexity of a vendors’ organisation increases component quality and quality of ongoing support reduces, thus contributing to increasing TCO.

### Concept Increasing effort
Comparing Functional redundancy ARiB3 with Increasing component integration effort ARiB7

There is nothing new between the comparisons of these 2 codes. Using components where only part of the functionality is required can require increasing effort for system developers to disable this functionality. Furthermore, increasing effort can also be required when maintaining systems in which component functionality has been disabled because, for example, upgrading the component can reactivate the redundant functionality, thus requiring additional effort to disable it again.

### Concept Balancing cost challenges
Comparing Functional redundancy ARiB3 with Component licensing fees ARiB11

It can be seen that system developers are balancing cost challenges by paying the whole licence fees of fewer large components but expending additional effort if disabling redundant functionality.

Concept **Balancing cost challenges**
Comparing Functional redundancy ARiB3 with Using fewer components ARiB18

From interview ARiB data using fewer components is expected to reduce system TCO. However, using fewer, larger components in a system can result in component functional redundancy when more functionality then required is provided. The result is system developers balancing cost challenges of reducing integration and ongoing costs by using fewer components with increasing costs of disabling redundant component functionality.

Concept **Conflicting design decisions**
Comparing Customer requesting component ARiB4 with Incorporating fewer components ARiB13

The choice for Incorporating fewer components in a system can be seen as a design decision aimed at reducing system complexity. However, when customers, rather than architects, choose components Conflicting design decisions may ensue because architects can be forced to implement an inappropriate system design comprising of additional components to accommodate customer selections.

Concept **Reducing user intervention**
Comparing Selecting Pre-integrated components ARiB9 with Reducing component installation effort ARiB10

Selecting pre-integrated components contributes to reducing integration effort by reducing user intervention (however, note that pre-integrated components may still need to be integrated with other parts of the system).

Concept **Balancing cost challenges**
Comparing Selecting Pre-integrated components ARiB9 with Increasing component licensing costs ARiB12

Selecting Pre-integrated components can reduce integration costs by saving on integration effort. However, pre-integrated components can incur additional licensing costs. Therefore, system developers, when choosing pre-integrated components, are balancing cost challenges of reducing integration effort with increasing component licensing costs.

Concept **Design decision**
Comparing Selecting Pre-integrated components ARiB9 with Incorporating fewer components ARiB13

Selecting Pre-integrated components can be seen as a Design decision, made enabling system designers to incorporate fewer component units into a system because each pre-integrated component set can be viewed as a single component unit.

Concept **Cost reducing strategy**
Comparing Selecting Pre-integrated components ARiB9 with Lowering total system costs ARiB13
Selecting Pre-integrated components can be seen as a **cost reducing strategy** employed by system designers to save integration and maintenance effort, thus contributing to Lowering total system costs.

**Concept Design decision and balancing cost challenges**
Comparing Reducing component installation effort ARiB10 with Increasing component licensing costs ARiB12

The choice by system developers to select pre-integrated components can be seen to be a **design decision** aimed at reducing component installation effort at the price of increasing licensing costs. However, designers are **balancing cost challenges** by acknowledging that increasing component licensing costs, which can outweigh the savings in installation effort, are preferable to the increased effort of integrating and maintaining separate components.

**Concept Cost reducing strategy**
Comparing Reducing component installation effort ARiB10 with Incorporating fewer components ARiB13

Building systems from the least number of COTS components can be seen to be a cost reducing strategy employed by system developers, aimed at reducing component installation effort, because fewer components require less integration effort to make the components work together (integration effort can involve glue code production, writing of integration scripts etc.).

**Concept Design decision and cost reducing strategy**
Comparing Incorporating fewer components ARiB13 with Lowering total system costs ARiB13

The decision by system developers for Incorporating fewer components into COTS-based systems can be seen to be both a **design decision** (because using fewer components is deemed beneficial to simplifying system design by reducing the number of interconnections) and a **cost reducing strategy** because of the following reasons: with fewer components there is less integration effort, with fewer components the effort required to manage vendor relationships can be reduced and with fewer components there is reduced exposure to forced changes to components, and thus the effort to address change.

**Concept Balancing design principles**
Comparing Homogeneity of vendors ARiB14 with Vendor’s organisational complexity ARiB15

In interview A (PM) the concept Homogeneity of vendors was seen as being beneficial to CBD because the assumption by system developers was that components supplied by the same vendor would support the same architectural standard, requiring less effort to integrate and thus contributing to reducing costs. However, from Interview B (Architect) Homogeneity of vendors is not necessary a cost reducing factor, especially if the vendor’s organisational structure is complex. There can appear little synergy between internal departments resulting in poor support and cooperation for customers attempting to integrate and maintain components supplied by the same vendor. In some cases, different departments appear as separate companies to the customer. Thus the service can be as bad as when dealing with different companies. Therefore, the assumption that vendor homogeneity is a beneficial design decision, because components supplied by the same vendor require less integration and maintenance effort, is contradicted if the vendor’s organisation is too complex to
provide good customer support. Therefore, in this case system developers are balancing design principles with the benefits of vendor homogeneity and deficiencies of complex vendor organisations.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Beyond the sphere of influence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparing Vendor commitment ARiB23 with Vendor support quality ARiB24</td>
<td></td>
</tr>
</tbody>
</table>

Vendor commitment to providing product support is crucial to system developers/maintainers receiving adequate vendor support.

Both vendor commitment and vendor support quality can be beyond the sphere of influence of system developers if vendors fail to commit to working closely with clients to resolve bugs, help resolve integration issues etc. For example, vendors supplying components to a small customer base may appear disinterested in improving customer care to encourage customers to migrate to different components.

Furthermore, vendor support quality can be a consequence of a vendor’s commitment to work with the client to resolve any bugs, help resolve integration issues etc. – i.e. a lack of vendor commitment can result in poor vendor support quality.

Can vendor commitment to working with clients be related to Vendor’s business culture?

| Comparing codes to previously generated concepts |
|-----------------|-----------------|
| Concept Increasing Effort and Losing faith |
| Comparing code: Poor component quality ARiB2 |

Poor component quality can lead to concept Increasing effort because system developers/maintainers expend increasing amounts of effort when continuously liaising with vendors to get bugs fixed, applying patches, testing and dealing with disruptions to system operation.

Continuous poor component quality can also contribute to system developers Losing faith with vendor’s ability to provide quality products, especially when components continuously fail due to bugs etc. If more than one component is considered as being of poor quality then this is reflected in system developers/maintainers holding negative views of vendors’ organisation.

Some components are described as being of poor quality because they do not work as intended and often display bugs. If this pattern occurs with more than one of the vendor’s components

| Concept Increasing costs |
|-----------------|-----------------|
| Comparing code: Functional redundancy ARiB3 |

Functional redundancy can result in increasing costs for system developers because disabling redundant functionality requires additional effort, which may need to be repeated following component upgrades.

| Concept Design decision |
|-----------------|-----------------|
| Comparing code: Customer requesting component ARiB4, ARiB5 |

Customer requesting component can still be seen as a Design decision because the component contributes to the overall system design. However, the difference is that
this type of design decision is not necessarily made or supported by system architects. Furthermore, this type of design decision can affect the overall system design because changes in system architecture may have to be made to accommodate it.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Design constraint – note changed to CONFLICTING DESIGN PRINCIPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Comparing code: Requiring external certification ARiB4</td>
</tr>
</tbody>
</table>

The selection of components because they’re externally certified can be seen as a **Design constraint** if they were forced upon system developers and do not fit in with the architectural basis of the rest of the system.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Increasing effort</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Comparing code: Integrating components ARiB6</td>
</tr>
</tbody>
</table>

Integrating components is linked to the concept of **increasing effort** for system developers because integrating components with other components or system parts requires human effort. Therefore, integrating more components incurs increasing effort.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Increasing effort</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Comparing code: Increasing component integration effort ARiB7</td>
</tr>
</tbody>
</table>

Increasing component integration effort can be seen to be a property of the concept **increasing effort** because increasing component integration effort is a specific example of increasing effort.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Balancing cost challenges</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Comparing code: Vendor supplying upgraded components ARiB8</td>
</tr>
</tbody>
</table>

Vendor supplying upgraded components can be seen to be linked to concept **Balancing cost challenges** because on one hand system developers save maintenance costs by accepting upgraded components from vendors; the vendors perform the upgrading tasks. However, on the other hand system developers/maintainers can incur additional costs when re-integrating the upgraded components into the system – in some instances the original integration tasks have to be redone. Therefore, system developers are balancing the maintenance cost savings of vendors providing upgrade components with maintenance cost additions of reintegrating upgraded components into the system.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Cost reducing strategy and Balancing cost challenges and reducing system complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Comparing code: Selecting Pre-integrated components ARiB9</td>
</tr>
</tbody>
</table>

Selecting Pre-integrated components can be seen to be a **cost reducing strategy** employed by system developers because pre-integrated components require less integration effort to deploy compared with the effort required to integrate the separate parts. Selecting Pre-integrated components can also be linked to **Balancing cost challenges** because although the integration effort for integrating pre-integrated may be less than integrating separate components the licensing costs of acquiring pre-integrated components can be higher than the licensing costs of the individual components. So, by saving integration effort system developers may incur additional licensing costs. However, as noted in the data the additional licence cost is far preferable to the increased effort of integrating separate components because it is predictable. Selecting Pre-integrated components can also contribute to **reducing system** complexity.
**complexity** because the if the vendor handles the integration and the ongoing maintenance of a pre-integrated ‘set of components’ – (a pre-integrated component collection can be viewed as a single entity – system complexity can be measured by the number of separate connections managed by system developers/maintainers) system developers only have to worry about integrating this entity with the remaining parts of the system, thus reducing the number of connected components they have to manage (thus a consequence of Selecting Pre-integrated components can be **reducing system complexity**) – note: ‘processing complexity’ may not be affected.

**Concept Strategy for reducing costs**
Comparing code: Removing human intervention ARiB10
Removing human intervention can be seen to be property of component **strategy for reducing costs**

The choice by system developers of **Removing human intervention**, by selecting components which are installed silently without user intervention can be seen as a **strategy for reducing costs** because removing the requirement for installation effort contributes to reducing some of the system integration costs.

**Concept Increasing costs**
Comparing code: Component licensing fees ARiB11
Component licensing fees can be seen to be a contributing factor to **increasing costs** for system developers because COTS component vendors tend to produce components to sell for a profit – component licensing fees are one way of a vendor charging for their products.

**Concept Balancing cost challenges**
Comparing code: Increasing component licensing costs ARiB12

System developers are **balancing cost challenges** of developing COTS-based systems by offsetting the integration cost savings of selecting pre-integrated components with **Increasing component licensing costs**. However, the interview data the “additional licensing costs are far preferable to the increased effort of integrating individual components.”

**Concept Design decision, reducing system complexity**
Comparing code: Incorporating fewer components ARiB13

The choice for **Incorporating fewer components** in a COTS-based system can be seen as a **design decision** made by system developers, believing that this contributes to reducing system total cost of ownership.

Incorporating fewer components can also be seen to contribute to **reducing system complexity** because by using fewer components the numbers of interconnections are reduced. This also reduces the number of interfaces, between components, to configure and maintain. System complexity is measured as the number of connected parts rather than the application processing complexity running on the system. This decision is determined because COTS components can be considered as black boxes rendering the component’s inner working transparent to system developers. System complexity, from system developer’s perspective relates to configuration actions (glue code, configuration scripts, component interface configuration etc.) they perform to enable individual components to work together.

Incorporating fewer components can also

**Concept Design decision**
Comparing code: Homogeneity of vendors ARiB14
The significance of Homogeneity of vendors is that components supplied by the same vendor support the same architectural standard and thus are easier to integrate than a set of components supplied by different vendors which do not support the same standard. Therefore, for this reason selecting components from the same vendor can be seen as a **design decision** made by system developers.
Appendix B3 - Interview with IT Architect (ARiC)

Contact summary form – field notes from interview:

<table>
<thead>
<tr>
<th>Contact Summary Form</th>
<th>Interview C (iC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contact name &amp; position:</td>
<td>, IBM Global Business Services, IT Architect</td>
</tr>
<tr>
<td>Contact Type: Interview, Document etc.:</td>
<td>Interview</td>
</tr>
<tr>
<td>Contact Date:</td>
<td>IBM North Harbour, Portsmouth</td>
</tr>
<tr>
<td>Visit:</td>
<td>X</td>
</tr>
<tr>
<td>Phone:</td>
<td></td>
</tr>
</tbody>
</table>

**NOTES**

1. Main issues and themes discussed: Cost Factors of COTS-based development

2. Summary of information gathered (of failed to collect) on each target question:

3. Other interesting points:

4. What further information required?

System dependencies increase maintenance costs. For example, COTS components may only work on certain OS levels and require specific database levels. Thus, a seemingly small change to a component or OS can have major implications as many other components and system parts have to be upgraded.

System requirements always change over life of system. Small changes with CBS will be cost more than in a custom environment because the costs of COTS implementation and tailoring tend to be higher than with custom systems.

Change is disruptive. There’s also a domino effect. For example, if a component which has reached its end of life needs replacing all of the initial integration effort may have to be redone. The other components may also have to be replaced because they won’t work with the new component – resulting in the whole initial implementation being redone.

In his experience upgrading of components tend to be grouped in order to reduce testing costs. If vendors don’t have enough customers they’ll withdraw a component. If it’s a crucial component to your business then you’ve a problem!

To reduce ongoing costs one recommendation is to build systems from fewer components, ideally supplied from same vendor (check that the components are developed by the vendor and that the vendor has not just bought the company!) and pre-integrated because the main costs are getting and keeping them working together.

Some customers have a firm principle to use COTS components regardless of whether they will support all requirements; they have the expectation that they’ll get a suitable system and the cost savings, which is wishful thinking.

With cost bear in mind that change occurs over time. For example, over 10 years the OS, hardware and requirements would have changed so the cost benefits of CBS do not last long.

**Open coding:** interview open coding

<table>
<thead>
<tr>
<th>CodeNo.</th>
<th>Key Text</th>
<th>Early Open Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARiC1</td>
<td>System dependencies increase maintenance costs.</td>
<td>System dependencies affecting maintenance costs</td>
</tr>
<tr>
<td>CodeNo.</td>
<td>Key Text</td>
<td>Early Open Codes</td>
</tr>
<tr>
<td>--------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>ARiC2</td>
<td>a seemingly small change to a component or OS can have major implications</td>
<td>Implications of change</td>
</tr>
<tr>
<td>ARiC3</td>
<td>as many other components and system parts have to be upgraded.</td>
<td>Knock-on effect</td>
</tr>
<tr>
<td>ARiC4</td>
<td>System requirements always change over life of system</td>
<td>Requirements changing over time</td>
</tr>
<tr>
<td>ARiC5</td>
<td>Small changes with CBS will cost more than in a custom environment</td>
<td>Underestimating CBS change costs</td>
</tr>
<tr>
<td>ARiC6</td>
<td>because the costs of COTS implementation and tailoring tend to be higher than with custom systems</td>
<td>Attributing higher re-integrating costs</td>
</tr>
<tr>
<td>ARiC7</td>
<td>Change is disruptive</td>
<td>Change causing disruption</td>
</tr>
<tr>
<td>ARiC8</td>
<td>There’s also a domino effect</td>
<td>Domino effect</td>
</tr>
<tr>
<td>ARiC9</td>
<td>For example, if a component which has reached its end of life needs replacing all of the initial integration effort may have to be redone</td>
<td>Redoing integration work</td>
</tr>
<tr>
<td>ARiC10</td>
<td>The other components may also have to be replaced because they won’t work with the new component</td>
<td>Replacing other components</td>
</tr>
<tr>
<td>ARiC11</td>
<td>resulting in the whole initial implementation being redone</td>
<td>Redoing initial implementation</td>
</tr>
<tr>
<td>ARiC12</td>
<td>upgrading of components tend to be grouped in order to reduce testing costs</td>
<td>Grouping components for upgrading</td>
</tr>
<tr>
<td>ARiC13</td>
<td>If vendors don’t have enough customers they’ll withdraw a component</td>
<td>Commercial viability of components</td>
</tr>
<tr>
<td>ARiC14</td>
<td>If it’s a crucial component to your business then you’ve a problem!</td>
<td>Attributing importance of business critical components</td>
</tr>
<tr>
<td>ARiC15</td>
<td>To reduce ongoing costs one recommendation is to build systems from fewer components</td>
<td>Reducing ongoing costs by selecting fewer components</td>
</tr>
<tr>
<td>ARiC16</td>
<td>ideally supplied from same vendor (check that the components are developed by the vendor and that the vendor has not just bought the company!)</td>
<td>Choosing components supplied by same vendor</td>
</tr>
<tr>
<td>ARiC17</td>
<td>and pre-integrated because the main costs are getting and keeping them working together.</td>
<td>Choosing pre-integrated components</td>
</tr>
<tr>
<td>ARiC18</td>
<td>Some customers have a firm principle to use COTS components regardless of whether they will support all requirements</td>
<td>Customer pre-conception</td>
</tr>
<tr>
<td>ARiC19</td>
<td>they have the expectation that they’ll get a suitable system and the cost savings, which is wishful thinking</td>
<td>Customer expectation</td>
</tr>
<tr>
<td>ARiC20</td>
<td>With cost bear in mind that change occurs over time</td>
<td>Linking cost with change over time</td>
</tr>
</tbody>
</table>
**Concept Degree of dependency**
Comparing System dependencies affecting maintenance costs ARiC1 and Implications of change ARiC2

System dependencies affecting maintenance cost explains that there can be an association between dependent COTS-based system parts and the effect on cost for system maintainers when maintaining these systems. The assumption by system maintainers is that more system dependencies can result in higher maintenance costs because performing maintenance on one system part can require other system parts to also be maintained. Maintaining normally means changing something.

Implications of change explain that change can have different implications.

The link between the two codes is that seemingly small changes to any parts of COTS-based systems can have major implications to the amount of effort required by system maintainers to effect the changes. For example, system dependencies may require component A to be installed on a specific operating system level with middleware products at specific levels. In turn, other COTS components may only work if component A is at a specific level. Therefore, a change to the operating system may require component A to be upgraded, which in turn requires the middleware and other system components to also be changed.

There are 3 factors involved in comparing the 2 codes: System dependencies, maintenance costs and change. It can be seen that the degree of dependency is key to explaining the relationship because the implications of change is related to the degree of dependency (because if the degree of dependency is high then the implications of change can be higher because changing one entity also requires other entities to be changed). Maintenance costs can be seen to be the consequence of the degree of dependency because the amount of maintenance effort, and thus cost, is linked to the amount of change required.

Degree of dependency can also be seen to a contributing factor of system complexity because as the degree of dependency increases the number of factors to be considered when developing and maintaining a system tends to increase.

**Concept Degree of dependency**
Comparing System dependencies affecting maintenance costs ARiC1 and Knock-on effect ARiC3

The concept degree of dependency links these 2 codes because when deciding to upgrade one component the degree of system dependencies can necessitate a ‘knock-on effect’ of system maintainers needing to upgrade other components in order to preserve system stability.

Concept Designing for change
Comparing System dependencies affecting maintenance costs ARiC1 and Requirements changing over time ARiC4

The basis of code Requirements changing over time is that system requirements
normally always change over time. Therefore, assuming maintenance costs are affected by system dependencies then its effect, in conjunction with constantly changing system requirements, would likely be greater on the resulting maintenance costs, compared with more stable systems with fewer dependencies. I.E. systems with greater system dependencies, which are also changing, require more effort to maintain because system maintainers need to consider more interrelated factors.

Therefore, system designers should employ the concept designing for change when designing CBS to reduce system dependencies and to facilitate system requirements changing over time because enabling systems to be changed easily (whether as part of maintenance or changing system requirements) should reduce ongoing costs.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Increasing cost</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Comparing Implications of change ARiC2 and Knock on effect ARiC3 25JAN09</strong></td>
<td></td>
</tr>
<tr>
<td>The implications of change on COTS-based systems can be the ‘knock on effect’ of further change because not all entities can be changed in isolation of other system components. The result can be <strong>increasing cost</strong> for system maintainers because additional effort is required in assessing dependencies, performing additional change activity and system testing. Thus, a seemingly small change can result in more change activity because of the underlying dependencies.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Concept</th>
<th>Change unpredictability</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Comparing Implications of change ARiC2 and Requirements changing over time ARiC4</strong></td>
<td></td>
</tr>
<tr>
<td>From the interview data ‘change’ is a major cost driver for COTS-based system developers and maintainers because it requires human effort. Depending upon the nature of the change activity the requirement may be for people with different skill types, including consultancy and other specialist skills, which may incur greater cost. Change influences cost because it can be unpredictable.</td>
<td></td>
</tr>
</tbody>
</table>

The link between codes Implications of change ARiC2 and Requirements changing over time ARiC4 identifies this. *The Implications of change* can be unpredictability because a change to one part of a system can necessitate further changes to other system parts due to system dependencies. A knock-on-effect of subsequent change requirements cannot always be predicted.

*Requirements changing over time* relates to system requirements changing throughout the life of a system. The interview data indicates that system requirements normally always change over the system’s lifespan because the business requirements of the system’s owner change. Changing business requirements, which are not always predictable when companies are exploiting new business opportunities, can necessitate changing system requirements.

Therefore, codes Implications of change ARiC2 and Requirements changing over time ARiC4 can be seen to be associated to the concept Change unpredictability, because the implications of change can be unpredictability and requirements changing over time can introduce change unpredictability. Change unpredictability can contribute to increasing cost for system maintainers. The act of ‘Change’ adds cost because it requires effort to assess, initiate and test the change. If change is planned cost can be minimised because the change complexity can be assessed in advance, the right skills brought in and people trained. With unpredictable change there may not be time to train people, thus skilled people may need to be brought in at short notice, which can cost more.
## Concept **Implications of change**

**Comparing** Implications of change ARiC2 and Underestimating COTS-based system change costs ARiC5

**Modified memo**

From the interview data Underestimating COTS-based system change costs ARiC5 by system practitioners is common because changes affecting COTS-based systems can not always be predicted or planned for. Furthermore, changes affecting seemingly distinct parts of COTS-based systems cannot always be performed in isolation of other work. Thus, small changes to one part of a system can require additional changes to other parts of the system. The consequence is that a seemingly small, inexpensive piece of work turns into a much larger work stream incurring much higher cost than initially envisaged. For example, changing one part of a COTS-based system can disrupt previously performed integration work necessitating this work to be redone.

Therefore, for system practitioners Underestimating COTS-based system change costs ARiC5 can be seen to be part of the wider theme, Implications of change ARiC2. This statement implies that there may be other properties of the Implications of change which have not yet emerged from the data.

++++++++++++++++++++++++++++++++++++++++++++++++

## Concept **Understanding implications of change**

**Comparing** Implications of change ARiC2 and Underestimating CBS change costs ARiC5

From the interview data Underestimating CBS change costs by system developers is normal because changes to COTS-based systems tend not to just involve actioning the change in isolation of other work, such as redoing integration tasks. Changing one part of a COTS-based system can disrupt previously performed integration work, upgrading a component can wipe configuration settings. Therefore, system developers are underestimating CBS change costs if they do not appreciate the implications of change – a small change to one part of a system can require changes to other parts of the system.

Both codes are related to the concept of system developers Understanding implications of change because to manage cost they need an understanding of the implications of changing CBS to avoid Underestimating CBS change costs. Thus failing to understand implications of change can result in increasing costs for system developers.

System developers focusing upon can be seen as a strategy for reducing costs

## Concept **Designing for change**

**Comparing** Implications of change ARiC2 and Change causing disruption ARiC7

The basis of code Change causing disruption ARiC7 is that change can be disruptive to system operations. For example, applications may have to be brought down when applying operating system patches, change can cause problems in the production system if not adequately tested, thus requiring a change to be backed out. Change causing disruption can add cost for system maintainers if change has to be performed outside of business hours because staff wages can be more during these times.
From a cost perspective, controlling the effect of **Implications of change** and **Change causing disruption**, for system maintainers, it is desirable for system designers, when designing systems, to implement the concept of **designing for change** because enabling one part of a system to be changed without the knock-on effect of other changes can reduce overall change activity, change complexity, amount of change disruption and thus, the cost of change. It is important for designing for change to be implemented at the system design phase because it is not normally possible to retrospectively modify the design of a system once it has been implemented.

**Concept Change unpredictability**
Comparing Implications of change ARiC2 and Domino effect ARiC8

Codes Implications of change ARiC2 and Domino effect ARiC8 can be seen to be linked to the concept of **change unpredictability** because of system dependencies the implications of changing one part of a system can result in the domino effect of other system parts also requiring changing – this series of changes cannot always be predicted by system developers/maintainers when the system was originally developed.

**Concept Redoing integration work**
Comparing Implications of change ARiC2 and Redoing integration work ARiC9

It can be seen from the data that one of the implication of change to COTS-based systems can be a requirement for system developers/maintainers **Redoing integration work** because making changes to COTS components, such as upgrading a component, can necessitate the rewriting of integration code if the vendor has changed the way the upgraded component works. **Redoing integration work** therefore adds cost for COTS-based system maintainers because of the human effort required in producing integration code.

**Concept Knock-on-effect**
Comparing Implications of change ARiC2 and Replacing other components ARiC10

The link between codes Implications of change ARiC2 and Replacing other components ARiC10 can be seen to be the concept of **Knock-on-effect** because the implications of changing one component can result in the **Knock-on-effect**, for system maintainers, of replacing other components due to architectural factors or other dependencies.

The concept **Knock-on-effect** is linked to increasing costs because it implies additional effort for system developers/maintainers. For example, system maintainer’s effort estimation for changing one component can be increased if the task turns out to involve replacing other components.

The concept of Knock-on-effect is not related only to replacing other components. The implications of change on COTS-based systems can have different knock-on-effects: redoing integration tasks…..

**Concept Knock-on-effect**
Comparing Implications of change ARiC2 and Redoing initial implementation ARiC11

Codes Implications of change ARiC2 and Redoing initial implementation ARiC11 are linked to concept **Knock-on-effect** because……...

**Concept Strategy for reducing costs**
Comparing Grouping components for upgrading ARiC12 and Reducing testing costs ARiC11

Codes Grouping components for upgrading ARiC12 and Reducing testing costs ARiC12 are linked to concept system maintainer’s **strategy for reducing costs** because upgrading a set of components in a group can lead to reducing testing costs because all of the upgraded components can be tested at the same time.

Upgrading can be disruptive, if it involves applications to be shut down. Therefore, for critical applications finding change windows can be difficult for change management – grouping components for upgrade can assist in scheduling change activity and can be preferable to upgrading each component separately at different times.

Concept **Conflicting business motives**

Comparing Commercial viability of components ARiC13 and Attributing importance of business critical components ARiC14

Codes Commercial viability of components ARiC13 and Attributing importance of business critical components ARiC14 can be seen to be linked to the concept of **conflicting business motives** because there seems to be a conflict in the business motives of COTS component vendors, compared with the buyers (system developers) of their components. COTS component vendors may their profit by developing and selling components to customers. If a vendor does not have a sufficient customer base for the component they may withdraw it (keeping a component on the market requires investment on their part. They may need to fix problems, produce patches, evolve it etc.). System developers, on the other hand, incorporate components into a system to solve business problems – system developers make their profit from the system they develop (either by developing a system on commission or selling the system etc.). Therefore, if vendors withdraw components which are critical to the functioning of the system this causes problems for the system developer.

Concept **Strategy for reducing costs**

Comparing Selecting fewer components ARiC15 and Reducing ongoing costs ARiC15

The assumption is that by selecting fewer components system developers are selecting fewer, larger components to satisfy system requirements, rather than using a larger number of smaller components to build COTS-based systems. This can be viewed as a factor for reducing ongoing costs of managing the system because system developers believe that this is factor in reducing system complexity.

It can be seen that codes Selecting fewer components ARiC15 and Reducing ongoing costs ARiC15 can be linked to concept **strategy for reducing costs** if system developers make the conscious choice to build a system from fewer components in order to reduce ongoing costs.

Concept **Design decision**

Comparing Selecting fewer components ARiC15 and Choosing components supplied by same vendor ARiC16

Codes Selecting fewer components ARiC15 and Choosing components supplied by same vendor ARiC16 can be linked to several concepts:

**Design decision**: System developers *Selecting fewer components* and *Choosing*
**components supplied by same vendor** can be seen to be design decisions because using fewer components can be considered to reducing system complexity. System developers assume that components supplied by the same vendor support the same architectural standard, thus easing component integration.

**Reducing system complexity:** *Selecting fewer components* links to reducing system complexity because it contributes to reducing the number of connections between components. *Choosing components supplied by same vendor* implies selecting components which are built to integrate with each other because they support the same architectural standard.

<table>
<thead>
<tr>
<th>Concept <strong>Strategy for reducing costs</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparing Reducing ongoing costs ARiC15 with Choosing components supplied by same vendor ARiC17</td>
</tr>
</tbody>
</table>

The link between code Reducing ongoing costs ARiC15 and Choosing components supplied by same vendor ARiC17 can be linked to concept **strategy for reducing costs** because the assumption that components supplied by same vendor support the same architectural standard and are built to integrate together require less effort to integrate and to maintain. Therefore, with this in mind system developers, when selecting components supplied by same vendor are employing a strategy for reducing costs: integration costs and ongoing maintenance costs.

<table>
<thead>
<tr>
<th>Concept <strong>Reducing user intervention</strong> and <strong>strategy for reducing costs</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparing Reducing ongoing costs ARiC15 with Choosing pre-integrated components ARiC17</td>
</tr>
</tbody>
</table>

Code Choosing pre-integrated components can contribute to Reducing ongoing costs by **Reducing user intervention**, thus reducing human effort. Choosing pre-integrated components can also be seen as a **strategy for reducing costs** employed by system developers by reducing human effort to integrate and maintain the set of components.

<table>
<thead>
<tr>
<th>Concept <strong>balancing design principles</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparing Reducing ongoing costs ARiC15 with Customer pre-conception ARiC18</td>
</tr>
</tbody>
</table>

Some customers have a pre-conception to build systems using the COTS-based approach regardless of whether COTS components are suitable or satisfy the requirements. It can be seen that system developers are **balancing design principles** by designing systems which may not provide best architectural solution but satisfy the customer’s instructions.

<table>
<thead>
<tr>
<th>Concept <strong>conflicting design principles</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparing Reducing ongoing costs ARiC15 with Customer expectation ARiC19</td>
</tr>
</tbody>
</table>

If customers insist on using COTS solution when not suitable the ideal system design (built for maintainability, change etc.) may not be able to be produced. As such, customers may not realise the perceived cost savings of adopting the COTS-based approach. Therefore, the link between the 2 codes can be seen to be system developers dealing with **conflicting design principles** because when producing a design to satisfy customer expectations for using COTS the result may not lead to reducing ongoing costs.

There is therefore a requirement for managing customer expectations.

<table>
<thead>
<tr>
<th>Concept <strong>balancing cost challenges</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparing Reducing ongoing costs ARiC15 with Linking cost with change over time</td>
</tr>
</tbody>
</table>
ARiC20

There is a conflict between codes Reducing ongoing costs ARiC15 with Linking cost with change over time ARiC20 because changing COTS-based systems contributes to increasing ongoing cost, rather than to reducing costs. COTS-based systems change over time so the cost benefits of CBD do not last long. Therefore, system developers have the constant battle of **balancing cost challenges** by aiming to minimise cost challenges relating to change with the implementing system design which realises the perceived benefits of reducing ongoing costs.

<table>
<thead>
<tr>
<th>Concept beyond sphere of influence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparing Choosing components supplied by same vendor ARiC16 with Everything changing over time ARiC21</td>
</tr>
</tbody>
</table>

The issue is that when system developers are totally reliant on components supplied from one vendor, which subsequently goes out of business. Everything changing over time can encompass vendors ceasing to trade as well as software and hardware changing. The link between the 2 codes is **beyond sphere of influence** because system developers can choose to select components from one vendor they’re unable to control if the vendor ceases trading.
Appendix B4 - Interview with Test Architect (ARiD)

Contact summary form – field notes from interview:

<table>
<thead>
<tr>
<th>Contact Summary Form</th>
<th>Interview D (iD)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Contact name &amp; position:</strong></td>
<td></td>
</tr>
<tr>
<td>Test Architect</td>
<td></td>
</tr>
<tr>
<td><strong>Contact Type: Interview, Document etc.:</strong></td>
<td></td>
</tr>
<tr>
<td>Face to Face Interview</td>
<td></td>
</tr>
<tr>
<td><strong>Contact Date:</strong></td>
<td></td>
</tr>
<tr>
<td>Visit: North Harbour Meeting room 5</td>
<td></td>
</tr>
<tr>
<td>Phone:</td>
<td></td>
</tr>
<tr>
<td><strong>NOTES</strong></td>
<td></td>
</tr>
</tbody>
</table>

1. Main issues and themes discussed:

Factors which affect the Total Cost of Ownership of a COTS-based system

2. Summary of information gathered (of failed to collect) on each target question:

Vendors Statement of Compatibility and good internal support structures important for reducing TCO.

3. Other interesting points:

4. What further information required?

---

Interview with Alistair Thomas, Test Architect

Question: What are the factors which affect the Total Cost of Ownership of a COTS-based system? How does the number of components used to build a CBS affect the TCO?

If possible, it is beneficial to use COTS components which are supplied by the same COTS vendor. This is because the vendor would have performed their own integration testing. Therefore, this saves time during system testing. We do not have to test the components but test them in relation to our environment.

The costs of the IDS system have been higher because we have been attempting to get a function to work which was not explicitly supported by the vendor – the ability to provide a high availability (HA) failover solution. In principle the SiteProtector HA solution should have worked because it was using some of the inbuilt functions of MS SQL Server. However, after wasting much effort the vendor could not determine exactly why this would not work. However, the Payplus failover solution also uses SGL Server as the database – this solution does work.

The IDS solution contains 3 main components – SiteProtector application (which includes the HIDS sensors, Event Collectors, Application Server), MS SQL Server and openssh (openssh used to tunnel traffic through the high level firewall ports – ISS have no statement that opensh will work with SiteProtector).

The SWIFT solution contains the 3 main components: SAG, SAA and WebSphere MQ. When planning this solution they identified early on that this combination of components and the chosen configuration should work.

Many component vendors now integrate several components under the covers with their product. For example, when installing the Hotscan product a version of Oracle is also installed. The end-user has no influence on this part of the installation and once installed
cannot alter the configuration of Oracle. However, in this case there is no requirement for a separate Oracle license agreement – this is included within the Hotscan licence.

MySql is another product which is automatically installed under the covers of other vendor’s products.

The main reason why the ISS solution required much more effort (both development and maintenance) than, say the SWIFT solution, was due to a lack of understanding of the product (with this configuration) by both the vendor and the IBM project team. It was difficult to get good training on the product. Several members of the IBM team attended some ISS run courses. However, these courses were not very good.

A further problem, which added extra cost, concerned the way that the ISS product was supported internally. The SiteProtector product was supported by the IBM Network Service Delivery (NSD) team (they had the relationship with the vendor – ISS). However, the W2K servers, which hosted the various parts of the system, were supported by members of the project team. The net effect of this was that when a problem was first reported there were numerous instances of ‘passing the buck’. No one team would immediately take responsibility for the problem, often stating that the problem was within the other teams area of responsibility. The NSD team was also suffering a serious staffing shortage. The fact that they supported many other projects resulted in some problems taking a long time to get fixed.

With respect to the CBS Functional Density Rule of Thumb the number of different components used to build a system was not the main factor affecting TCO. It was deemed more important to receive a ‘statement of compatibility’ from different vendors that their components will work together. For example, Microsoft will work with different vendors to help them develop components. MS will then ‘certify’ that these components will work with MS operating systems.

The vendor/client relationship is also key in reducing TCO. For example, ISS have a ‘request for change’ process. However, the speed at which changes get addressed by ISS depends upon the relationship (and size) of the customer to ISS. Due to customer demand, ISS did recently produce an API to enable SiteProtector to work with Remedy.

The other main TCO issues which have been experienced with the IDS solution are:

- Costs increase if the COTS customer attempts to mould a component/product to fit the desired solution rather than buying in the correct set of components.
- Licensing issues can force a company to go with one COTS solution rather than look for alternative (better) products.
- Costs also increase if the vendor’s upgrade policy does not match the customer’s internal maintenance policies.

Open coding: (Test Architect) interview open coding

<table>
<thead>
<tr>
<th>CodeNo.</th>
<th>Key Text</th>
<th>Early Open Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARiD1</td>
<td>beneficial to use COTS components which are supplied by the same COTS vendor</td>
<td>Sourcing components from same vendor</td>
</tr>
<tr>
<td>ARiD2</td>
<td>because the vendor would have performed their own integration testing</td>
<td>Vendor testing integration</td>
</tr>
<tr>
<td>ARiD3</td>
<td>saves time during system testing</td>
<td>Saving time</td>
</tr>
<tr>
<td>ARiD4</td>
<td>We do not have to test the components but test them in relation to our environment.</td>
<td>Saving testing effort</td>
</tr>
<tr>
<td>ARiD5</td>
<td>costs of the IDS system have been higher</td>
<td>Increasing costs</td>
</tr>
<tr>
<td>ARiD6</td>
<td>attempting to get a function to work which was not explicitly supported by the vendor</td>
<td>Implementing unsupported function</td>
</tr>
<tr>
<td>ARiD7</td>
<td>after wasting much effort the vendor could not determine exactly why this would not work</td>
<td>Wasting effort Vendor support issue</td>
</tr>
<tr>
<td>CodeNo.</td>
<td>Key Text</td>
<td>Early Open Codes</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>ARiD8</td>
<td>[The SWIFT solution contains the 3 main components: SAG, SAA and WebSphere MQ.] When planning this solution they identified early on that this combination of components and the chosen configuration should work</td>
<td>Planning integration</td>
</tr>
<tr>
<td>ARiD9</td>
<td>Many component vendors now integrate several components under the covers with their product.</td>
<td>Acquiring pre-integrated components</td>
</tr>
<tr>
<td>ARiD10</td>
<td>The end-user has no influence on this part of the installation</td>
<td>Removing human intervention</td>
</tr>
<tr>
<td>ARiD11</td>
<td>and once installed cannot alter the configuration of Oracle</td>
<td>Removing human intervention</td>
</tr>
<tr>
<td>ARiD12</td>
<td>However, in this case there is no requirement for a separate Oracle license agreement – this is included within the Hotscan licence</td>
<td>Saving licensing cost</td>
</tr>
<tr>
<td>ARiD13</td>
<td>MySql is another product which is automatically installed under the covers of other vendor’s products</td>
<td>Removing human intervention</td>
</tr>
<tr>
<td>ARiD14</td>
<td>The main reason why the ISS solution required much more effort (both development and maintenance) than, say the SWIFT solution, was due to a lack of understanding of the product (with this configuration) by both the vendor and the IBM project team.</td>
<td>Lacking product understanding</td>
</tr>
<tr>
<td>ARiD15</td>
<td>It was difficult to get good training on the product</td>
<td>Poor product training quality</td>
</tr>
<tr>
<td>ARiD16</td>
<td>A further problem, which added extra cost, concerned the way that the ISS product was supported internally</td>
<td>Internal product support issues</td>
</tr>
<tr>
<td>ARiD17</td>
<td>The SiteProtector product was supported by the IBM Network Service Delivery (NSD) team (they had the relationship with the vendor – ISS)</td>
<td>Differing product support arrangements</td>
</tr>
<tr>
<td>ARiD18</td>
<td>However, the W2K servers, which hosted the various parts of the system, were supported by members of the project team</td>
<td>Differing product support arrangements</td>
</tr>
<tr>
<td>ARiD19</td>
<td>The net effect of this was that when a problem was first reported there were numerous instances of ‘passing the buck’</td>
<td>‘buck passing’</td>
</tr>
<tr>
<td>ARiD20</td>
<td>No one team would immediately take responsibility for the problem, often stating that the problem was within the other teams area of responsibility</td>
<td>Failing to take responsibility</td>
</tr>
<tr>
<td>ARiD21</td>
<td>The NSD team was also suffering a serious staffing shortage.</td>
<td>Staffing shortage</td>
</tr>
<tr>
<td>ARiD22</td>
<td>The fact that they supported many other</td>
<td>Lacking resource</td>
</tr>
<tr>
<td>CodeNo.</td>
<td>Key Text</td>
<td>Early Open Codes</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>ARiD23</td>
<td>the number of different components used to build a system was not the main factor affecting TCO.</td>
<td>Identifying other cost factors</td>
</tr>
<tr>
<td>ARiD25</td>
<td>It was deemed more important to receive a ‘statement of compatibility’ from different vendors that their components will work together.</td>
<td>Statement of compatibility</td>
</tr>
<tr>
<td>ARiD26</td>
<td>For example, Microsoft will work with different vendors to help them develop components.</td>
<td>Developing vendor cooperation</td>
</tr>
<tr>
<td>ARiD27</td>
<td>MS will then ‘certify’ that these components will work with MS operating systems</td>
<td>Selecting certified components</td>
</tr>
<tr>
<td>ARiD28</td>
<td>The vendor/client relationship is also key in reducing TCO</td>
<td>Establishing vendor/client relationship</td>
</tr>
<tr>
<td>ARiD29</td>
<td>For example, ISS have a ‘request for change’ process</td>
<td>Vendor accepting change requests</td>
</tr>
<tr>
<td>ARiD30</td>
<td>However, the speed at which changes get addressed by ISS depends upon the relationship (and size) of the customer to ISS</td>
<td>Variable vendor response</td>
</tr>
<tr>
<td>ARiD31</td>
<td>Due to customer demand, ISS did recently produce an API to enable SiteProtector to work with Remedy</td>
<td>Responding to customer requests</td>
</tr>
<tr>
<td>ARiD32</td>
<td>The other main TCO issues which have been experienced with the IDS solution are:</td>
<td>Identifying other costs factors</td>
</tr>
<tr>
<td>ARiD33</td>
<td>Costs increase if the COTS customer attempts to mould a component/product to fit the desired solution rather than buying in the correct set of components</td>
<td>Increasing cost Modifying components</td>
</tr>
<tr>
<td>ARiD34</td>
<td>Licensing issues can force a company to go with one COTS solution rather than look for alternative (better) products.</td>
<td>Licensing costs</td>
</tr>
<tr>
<td>ARiD35</td>
<td>Costs also increase if the vendor’s upgrade policy does not match the customer’s internal maintenance policies</td>
<td>Synchronising maintenance schedules</td>
</tr>
</tbody>
</table>

Initial memos: Memos (Interview)

Concept **reducing integration effort**
Comparing Sourcing components from same vendor ARiD1 with Vendor testing integration ARiD2
Codes Sourcing components from same vendor ARiD1 and Vendor testing integration ARiD2 are linked to the concept of **reducing integration effort** for system developers because the vendor should have performed integration testing ensuring that their set of components integrate and work together. This saves the system developer from performing this testing – they just need to test that the components work within the system.

Concept **reducing integration effort**
Comparing Sourcing components from same vendor ARiD1 with Saving time ARiD3

See memo ‘Comparing Sourcing components from same vendor ARiD1 with Vendor testing integration ARiD2’ above as same conditions apply

Concept **reducing integration effort**

Comparing Sourcing components from same vendor ARiD1 with Saving testing effort ARiD4

See memo ‘Comparing Sourcing components from same vendor ARiD1 with Vendor testing integration ARiD2’ above as same conditions apply

Concept **reducing user intervention, Strategy for reducing costs and reducing system complexity**

Comparing Sourcing components from same vendor ARiD1 with Acquiring pre-integrated components ARiD9

Codes Sourcing components from same vendor ARiD1 and Acquiring pre-integrated components ARiD9 contributes to concept **reducing user intervention** because acquiring sets of pre-integrated components, supplied by the same vendor can requires less human effort when installing and upgrading the components.

Codes Sourcing components from same vendor ARiD1 and Acquiring pre-integrated components ARiD9 can also be seen to be linked to the concept **Strategy for reducing costs** because some system developers select pre-integrated components, supplied by the same vendor, in order to reduce the installation and maintenance effort, and thus cost, which could be incurred if installing a set of separate components.

It can also be seen that Codes Sourcing components from same vendor ARiD1 and Acquiring pre-integrated components ARiD9 contribute to **reducing system complexity** because with pre-integrated components there are fewer interfaces needing to be managed by system developers (the assumption is that the number of interconnected interfaces contributes to system complexity because each separate interface requires effort by system developers to connect, integrate and manage. With pre-integrated components the connection, integration and management effort of the set of components comes under the vendor’s responsibility – system developers only have to manage their integration with the rest of the system)

Concept **Beyond sphere of influence and increasing effort**

Comparing Lacking product understanding ARiD14 with Poor product training quality ARiD15

The link between codes Lacking product understanding ARiD14 with Poor product training quality ARiD15 was the concept **Beyond sphere of influence** for system developers.

There was a lack of product understanding by both the vendor and the company’s internal support team. Furthermore, the company’s internal support team were unable to get adequate training on the vendor’s components (the vendor ran educational classes on the components but they were described as poor quality by the attendees). The result, from the system developer’s perspective, was manifested by numerous problems with the components (software quality and configuration issues) during the system development and maintenance phases – these issues were **beyond the sphere of influence** of system developers because they had no influence over the internal
support team’s product knowledge or the product support quality provided by the component vendor.

It was suggested from the interview data (interview ARiD) that the organisational culture of the company resulted in internal departments behaving like separate companies; the consequence being a lack of cooperation between different departments.

It can be seen that codes Lacking product understanding ARiD14 with Poor product training quality ARiD15 also contributes to increasing costs because of increasing effort incurred by system developers/maintainers managing product problems, pursuing resolution to product problems from vendor and internal support teams and testing proposed solutions to numerous product problems.

Concept **Organisational concerns**
Comparing Lacking product understanding ARiD14 with Internal product support issues ARiD16

The link between codes Lacking product understanding ARiD14 with Internal product support issues ARiD16 is concept **Organisational concerns** for system developers/maintainers as they have concerns about the vendor and internal support team organisation’s ability to provide adequate product support. Therefore, system developers/maintainers are lacking confidence in both the vendor’s and internal support team’s ability to resolve problems with the components due to their lacking product understanding.

Concept **Losing faith**
Comparing Lacking product understanding ARiD14 with Differing product support arrangements ARiD17, ARiD18

It can be seen that the combination of codes Lacking product understanding ARiD14 with Differing product support arrangements ARiD17, ARiD18 results in system developers/maintainers Losing faith with the vendor’s and internal support team’s ability to support the components due to their poor product understanding and complicated internal product support structure.

Concept **Denying responsibility**
Comparing Internal product support issues ARiD16 with ‘buck passing’ ARiD19

The link between codes Internal product support issues ARiD16 and ‘buck passing’ ARiD19 is the concept of **denying responsibility** because by buck passing the company’s internal support groups were denying responsibility for the cause and resolution to problems with components, which contributed to internal product support issues for system developers/maintainers.

From the data **denying responsibility** appears to relate to the company’s internal culture because although the different support teams are part of the same company they behave as if separate company boundaries exist.
Comparing Failing to take responsibility ARiD17, ARiD18 with ‘buck passing’ ARiD19
As above in ‘Internal product support issues ARiD16 with ‘buck passing’ ARiD19’

Concept **Beyond sphere of influence**
Comparing Internal product support issues ARiD16 with Staffing shortage ARiD21

The link between codes Internal product support issues ARiD16 and Staffing shortage ARiD21 is **beyond sphere of influence** because system developers/maintainers have no control over the staffing levels of the company’s internal support teams.

Concept **Design objective** and **cost reducing strategy**
Comparing Identifying other cost factors ARiD23 with Statement of compatibility ARiD24

The link between codes Identifying other cost factors ARiD23 and Statement of compatibility ARiD24 is concept **design objective** because, from a system design perspective, vendor’s statement of compatibility gives system developers the confidence that different components will work together.
From a cost perspective, selecting components covered by a Statement of compatibility can also be seen to be a **cost reducing strategy** employed by system developers because components which have been assessed to work together are likely to cost less to integrate and maintain than components which do not have this guarantee.

Concept **Cost reducing strategy**
Comparing Saving testing effort ARiD4 with Statement of compatibility ARiD24

From a cost perspective, selecting components covered by a Statement of compatibility ARiD24 can be seen to be a **cost reducing strategy** employed by system developers because components which have been assessed to work together are believed to result in Saving testing effort ARiD4 compared with components which do not have this guarantee.

Concept **Establishing relationships**
Comparing Identifying other cost factors ARiD23 with Developing vendor cooperation ARiD25

The link between the 2 components is **establishing relationships** because the aim of system developers’ developing vendor cooperation is in persuading vendors to address concerns with components (such as making modifications to components etc.), which in turn can contribute to reducing integration and maintenance costs if the modifications facilitate easier component integration and maintenance.

**Establishing relationships** can also be seen to be a **cost reducing strategy** employed by system developers as building good working relationships (not just between system developer and vendor but between system developer and other development staff, support staff, management and customers) can contribute to reducing cost as people are more likely to cooperate if a good relationship exists.

Concept **Design objective**
Comparing Identifying other cost factors ARiD23 with Selecting certified components ARiD27

From a cost perspective, Selecting certified components can be seen to be related to the implementation of a **design objective** because by selecting certified components...
system developers gain assuredly that the components will work within specified operating environments. For example, from the interview data (ARiD) MS work with different vendors with the aim of developing and certifying components which will work with Windows operating systems.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Design objective</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Comparing Developing vendor cooperation ARiD26 with Establishing vendor/client relationship ARiD28</td>
</tr>
</tbody>
</table>

From the data the concept of **establishing relationships** appears key for system developers to facilitate the resolution of technical problems and service disagreements. Codes Developing vendor cooperation ARiD26 and Establishing vendor/client relationship ARiD28 are examples of **establishing relationships**

**Note:** look for evidence of maintaining relationships

<table>
<thead>
<tr>
<th>Concept</th>
<th>Beyond sphere of influence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Comparing Developing vendor cooperation ARiD26 with Variable vendor response ARiD30</td>
</tr>
</tbody>
</table>

When developing vendor cooperation system developers can be met with variable vendor response which can be **beyond their sphere of influence** because the speed of vendor’s response can be related to the perceived importance of the customer to the vendor, thus if the system developer’s organisation is not considered important to the vendor than then the quality of vendor cooperation could be less.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Component licensing fees and Balancing cost challenges</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Comparing Licensing costs ARiD34 existing component lists</td>
</tr>
</tbody>
</table>

Code Licensing costs can be related to concept **component licensing fees** because they are essentially the same thing. **Component licensing fees** tend to contribute to increasing costs for system developers because financial outlay is required. However, from the data (Interview ARiD), system developers are **balancing cost challenges** if choosing components with lower licensing costs, but requiring additional tailoring costs, than more functionally appropriate components with higher licensing costs.

<table>
<thead>
<tr>
<th>Concept</th>
<th>System maintenance complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Comparing Synchronising maintenance schedules ARiD35 existing component lists</td>
</tr>
</tbody>
</table>

Code Synchronising maintenance schedules contributes to **System maintenance complexity** because if the vendor’s upgrade policy does not match the system organisation’s internal maintenance policy more factors have to be considered by system maintainers when managing ongoing system maintenance and stability.
Appendix B5 - Interview with Software developer (SDiE)

Contact summary form – field notes from interview:

<table>
<thead>
<tr>
<th>Contact Summary Form</th>
<th>Interview E (iE)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Contact name &amp; position:</strong></td>
<td></td>
</tr>
<tr>
<td>– Software Developer/Support</td>
<td></td>
</tr>
<tr>
<td><strong>Contact Type: Interview, Document etc.:</strong></td>
<td></td>
</tr>
<tr>
<td>Face to face interview</td>
<td></td>
</tr>
<tr>
<td><strong>Contact Date:</strong></td>
<td></td>
</tr>
<tr>
<td>Visit: North Harbour 16:30 to 16:50</td>
<td></td>
</tr>
<tr>
<td><strong>Phone:</strong></td>
<td></td>
</tr>
</tbody>
</table>

1. **Main issues and themes discussed:**
The cost factors affecting systems built from COTs components

2. **Summary of information gathered (of failed to collect) on each target question:**

3. **Other interesting points:**

4. **What further information required?**

**Question:** What are the factors which affect the Total Cost of Ownership of a COTS-based system? How does the number of components used to build a CBS affect the TCO?

With regard to the CBS Functional density rule xxxx agrees that the number of components used within a system has an effect on the TCO of the system. This is directly related to the following factors:

1. Getting product support. The more different components used there are more people to talk to when requiring support. This is even the case if the products are supplied by the same vendor. Different products supplied by the same vendor are normally supported by different teams/departments within the vendor’s organisation. Very often this is like dealing with totally different companies.

2. The number of interfaces to develop and maintain. From a maintenance point of view it can be very difficult to tie a problem down. There is often much ‘passing of the buck’ where the vendor will deny that the problem is with their product and visa versa.

The one main factor to affecting the quality of a COTS product is the number of copies sold. XXXX suggests that the more copies of a product sold ultimately results in more people testing it!!

For example, 120 copies of Payplus have been sold. This is relatively low. xxxx thinks that this is one reason why there have been so many bugs with this product. Hotscan has more users. xxxx has noticed far fewer software bugs with this product.

With SAA (a SWIFT product), which is considered an industry leader within the banking arena there are a huge number of other companies using this product. XXXX has noticed no bugs with the SAA product.

AT this point XXXX was called to another meeting so had to leave.
## Open coding:  
**(Software Developer) interview open coding**

<table>
<thead>
<tr>
<th>CodeNo.</th>
<th>Key Text</th>
<th>Early Open Codes</th>
</tr>
</thead>
</table>
| SDiE1   | the number of components used within a system has an effect on the TCO of the system | Cost attributing factor  
Increasing number of components |
| SDiE2   | This is directly related to the following factors:                        | Linking factors                                        |
| SDiE3   | 1) Getting product support                                               |                                                       |
| SDiE4   | The more different components used there are more people to talk to when requiring support | Increasing number of components  
Increasing support complexity |
| SDiE5   | This is even the case if the products are supplied by the same vendor     | Vendor homogeneity                                      |
| SDiE6   | Different products supplied by the same vendor are normally supported by different teams/departments within the vendor’s organisation | Internal organisational complexity                     |
| SDiE7   | Very often this is like dealing with totally different companies.         | Lacking organisational synergy                         |
| SDiE8   | 2) The number of interfaces to develop and maintain                       | Developing additional interfaces  
Maintaining additional interfaces                        |
| SDiE9   | From a maintenance point of view it can be very difficult to tie a problem down. | Isolating problems                                     |
| SDiE10  | There is often much ‘passing of the buck’ where the vendor will deny that the problem is with their product and visa versa | Buck passing                                            |
| SDiE11  | The one main factor to affecting the quality of a COTS product is the number of copies sold. | Vendors selling more components  
Improving product quality                                 |
| SDiE12  | the more copies of a product sold ultimately results in more people testing it!! | Increasing test base                                   |
| SDiE13  | For example, 120 copies of Payplus have been sold. This is relatively low. xxxx thinks that this is one reason why there have been so many bugs with this product. Hotscan has more users. xxxx has noticed far fewer software bugs with this product. With SAA (a SWIFT product), which is considered an industry leader within the banking arena there are a huge number of other companies using this product. xxxx has noticed no bugs with the SAA product | Linking product quality  
with number of components sold                             |

**Initial memos:**  
Memos (Xxxx Xxxx - Software Developer - Interview)  

**Concept balancing cost challenges**  
Comparing Cost attributing factor SDiE1 with Linking factors SDiE2  
From the interview (interview SDiE) data there are cost attributing factors, such as
the number of components used in a COTS-based system. There are also *linking factors* providing reasons for the effect of the number of components used in systems on cost.

The choice to use a set number of components within a system normally results from design decisions made by system developers. However, when considering the number of components to be used in a design system developers may be **balancing cost challenges** arising from using more components with other costs which could occur if failing to support all system requirements or implementing the system on time.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Increasing support complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Comparing Increasing number of components SDiE1, SDiE4 with Increasing support complexity SDiE4</strong></td>
<td></td>
</tr>
</tbody>
</table>

The link between codes Increasing number of components SDiE1, SDiE4 and Increasing support complexity SDiE4 is the concept of *increasing support complexity* because a consequence of building systems from more components, supplied from different vendors or from different departments within the same vendor’s organisation, is the requirement for customers (system developers/maintainers) to consult with different sets of people when requiring support for different components making up the system. *Increasing support complexity* is defined in terms of the number of different vendor/customer support relationships required when supporting systems made up of many different components. Thus, a greater number of vendor/customer support relationships can lead to **increasing support complexity**.

From the interview data (SDiE) data **increasing support complexity** can be compounded for system developers/maintainers when performing problem determination and requiring product support from greater numbers of vendors or internal vendor departments at the same time. This is because in cases where there is difficulty in identifying the components or system parts causing the problem vendors may not be willing to take responsibility without evidence of their liability or to cooperate with each to resolve the problems.

A further examination suggests that the link between codes Increasing number of components SDiE1, SDiE4 with Increasing support complexity SDiE4 is a cause/consequence relationship because increasing support complexity can occur as a result of increasing number of components.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Increasing support complexity and <strong>balancing cost challenges</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Comparing Increasing number of components SDiE1, SDiE4 with Vendor homogeneity SDiE5</strong></td>
<td></td>
</tr>
</tbody>
</table>

From the data code **Vendor homogeneity SDiE5** encapsulates the notion of different components being supplied by the same vendor. From the data (previous interviews PMiA and ARiB) the choice of system developers to build COTS-based systems from components supplied by the same vendor was considered to be a design decision contributing to reducing costs because of the assumption that the components would be designed to integrate together. Thus, from a design perspective sourcing components from the same vendor is seen to reduce the effort required to tailor and integrate components supplied by different vendors. A further assumption from the interview data was that the maintenance effort would be less for components supplied
by the same vendor because vendor would test and verify the integration of upgraded components with all of their products.

Sourcing components from the same vendor was also deemed to reduce the development and maintenance costs of building a system from more components because of the assumption that the components were designed to integrate together.

However, from this data (Interview SDiE) large vendors producing many COTS components may comprise of a more complex internal organisational structure, whereby different components are developed and supported by different internal departments or teams. From a product support perspective the different departments of some large vendor organisations can appear to customers as virtually different companies, with little synergy and cooperation between departments. Thus, from the customer’s perspective the cost benefits of sourcing components from the same vendor can be challenged by an increasing support complexity manifested by the requirement of dealing with different support groups when needing product support.

Another link between codes Using more components SDiE1, SDiE4 and Vendor homogeneity SDiE5 can be concept balancing cost challenges because, on the one hand, building systems from more components can add cost for system developers due to the increasing cost of developing and maintaining more interfaces. Conversely, the cost of using a greater number of components can be offset by sourcing them from the same vendor because the assumption is that components supplied by the same vendor will integrate together with minimal effort. However, the cost benefit of sourcing the components could be diminished if the vendor’s internal support structure makes it difficult to gain adequate support for the variety of products.

Concept increasing support complexity and balancing cost challenges
Comparing Increasing number of components SDiE1, SDiE4 with Internal organisational complexity SDiE6
AND
Comparing Increasing number of components SDiE1, SDiE4 with Lacking organisational synergy SDiE7

Same as above: See memo ‘Comparing Increasing number of components SDiE1, SDiE4 with Vendor homogeneity SDiE5’

Concept Increasing effort and Conflicting design decisions
Comparing Increasing number of components SDiE1, SDiE4 with Developing additional interfaces SDiE8 and Maintaining additional interfaces SDiE8

The link between Increasing number of components SDiE1, SDiE4 and Developing additional interfaces SDiE8 can be seen to be concept Increasing effort because with more components more effort can be required to develop and maintain additional interfaces.

Comparing Increasing number of components SDiE1, SDiE4 with Developing additional interfaces SDiE8 and Maintaining additional interfaces SDiE8 can also be seen to be linked to system developers making Conflicting design decisions when selecting a design involving more components (which may satisfy more of the system requirements than an alternative design involving fewer components), but incurring additional integration and maintenance costs as a result of the additional interfaces.
required to connect a greater number of components together

Concept **increasing support complexity** and **attributing accountability**

Comparing Increasing number of components SDiE1, SDiE4 with Isolating problems SDiE9

From a support perspective, the number of components in a system affects the ability of system developers/maintainers and vendors to isolate problems.

When there is an interaction of different vendors’ products, in-house built glue code, wrapper code and possibly legacy application code **Isolating problems** can be challenging for customers (system developers/maintainers) and vendors. This can be compounded with systems containing more components because there may be more coding factors, such as interfaces, to consider as well as more customer/vendor relationships to manage. All of which can contribute to **increasing support complexity** when the customer has to deal with greater numbers of vendors’ support groups or different support teams within the same vendor organisation.

From the data the link between concept **increasing support complexity** and category **RELATIONSHIP COMPLEXITY** appears causal-consequence and degree because, for example, a consequence of a higher degree of RELATIONSHIP COMPLEXITY, brought about by dealing with greater numbers of vendor support groups, can be **increasing support complexity**, because the number of different people the customer has to talk to when isolating problems.

In complex environments the root cause of problems may not be obvious. This can be further compounded if vendors are denying responsibility for the problem because they feel the problem is occurring beyond the scope of their code [normally black box code]. Comparing complex systems comprising of more components, containing greater numbers of interfaces, glue code, wrapper code, **attributing accountability** may be one challenge of isolating problems because vendors may be unable to locate the root cause of a problem or unwilling to cooperate with other vendors (or other departments within the same vendor) to aid problem isolation. It could be suggested that **attributing accountability** of problems in systems with fewer components and integrated parts is more straightforward because there are less factors to consider.

However, from the interview data it is not just an Increasing number of components, linked to isolating problems, which affects **attributing accountability**, but the number of involved parties and complexities of the relationships between customer and vendors which may be important. For example, a system containing a greater number of components, possibly supplied by the same vendor, and where a good working relationship exists between customer and vendor and where the vendor willingly cooperates with the customer to aid problem resolution the challenge of **attributing accountability** by the customer may not exist because the vendor assumes a degree of responsibility.

Therefore, it appears that the link between codes Increasing number of components SDiE1, SDiE4 and Isolating problems SDiE9 and concept **attributing accountability** is a property of category **RELATIONSHIP COMPLEXITY** because the degree of relationship complexity may affect **attributing accountability**; where relationship complexity is reduced, such as in a case of only one customer and one vendor **attributing accountability** may be achieved and agreed more willingly than in cases.
where a customer has to deal with multiple vendors or multiple vendor departments.

**Concept denying responsibility**
Comparing Increasing number of components SDiE1, SDiE4 with Buck passing SDiE10

From the data (interview SDiE) there is often much ‘passing of the buck’ where the vendor will deny that the problem is with their product. The suggestion from the data is that with problems where the root cause is not easily attributable to a specific component and where increasing numbers of vendors are involved within problem diagnosis there may be a greater chance that vendors would be more willing to pass the buck if they could get a way with it. For example, in a situation where the system is built from only one component the vendor would have less justification for denying responsibility for any problems because their component in the only one making up the system.

When linking buck passing with an Increasing number of components there may be greater chances of vendors denying responsibility for problems, especially if the increasing number of components were supplied by different vendors because it may be increasingly difficult to tie a problem down to one component.

**Concept knock-on-effect**
Comparing Vendors selling more components SDiE11 with Improving product quality SDiE11, Increasing test base SDiE12 and Linking product quality with number of components sold SDiE13

From the data (interview SDiE) even if vendors test their components before releasing them for sale they are unlikely to identify all possible problems. Therefore, in order to improve the quality of their products vendors need to sell as many components as possible, to as wide a customer base as possible, thus enabling the products to be effectively tested within varied environments and implementations not considered during vendor testing. The larger and more varied the customer base the greater the chance of vendors being able to identify and fix a greater proportion of problems.

However, if after the initial release of the product significant problems are encountered and reported by the first customers to buy the components the knock-on-effect can be of dissuading other prospective customers from purchasing the products, thus denying the vendor the opportunity of improving product quality by testing within a larger customer base.
Appendix B6 - Interview with Application Architect (ARiF)

Contact summary form – field notes from interview:

<table>
<thead>
<tr>
<th>Contact Summary Form</th>
<th>Interview F (iF)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Contact name &amp; position:</strong></td>
<td>- Application Architect</td>
</tr>
<tr>
<td><strong>Contact Type:</strong> Interview, Document etc.:</td>
<td>Face to face interview</td>
</tr>
<tr>
<td><strong>Contact Date:</strong></td>
<td>Visit: North Harbour D1 café 15:00 to 16:00</td>
</tr>
<tr>
<td><strong>Phone:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>NOTES</strong></td>
<td></td>
</tr>
<tr>
<td>1. Main issues and themes discussed:</td>
<td>The cost factors affecting systems built from COTs components</td>
</tr>
<tr>
<td>2. Summary of information gathered (of failed to collect) on each target question:</td>
<td>There is no real ideal number of components for a COTS solution – this dependent on the required functionality of the system – however, the fewer components the better, in terms of complexity.</td>
</tr>
<tr>
<td>3. Other interesting points:</td>
<td></td>
</tr>
<tr>
<td>4. What further information required?</td>
<td></td>
</tr>
</tbody>
</table>

Question: What are the factors which affect the Total Cost of Ownership of a COTS-based system? How does the number of components used to build a CBS affect the TCO?

Websphere is designed around a common architecture. As such, each Websphere component has an MQ interface (MQ is used to send messages between different components and applications). COTS components supplied by different vendors are more of a factor affecting total system life costs than just the number of different components. One main issue is the number of dependencies different vendor’s components have – for example, when upgrading one component (term package, product and component used interchangeably) other vendors’ packages may also need to be upgraded.

Merva, an IBM produced payment package, originally worked with X25 network connections. SWIFT moved their communication protocol method from X25 to the IP based Swift Network Link (SNL). Therefore, this led to the move to Swift Alliance (SAA).

Main problems with the maintenance of COTS systems, which has had big influences on the life costs:
1. Keeping each product in support when different products have different life spans.
2. Hidden dependencies - It was not always obvious that certain dependencies existed until the requirement to upgrade another product.

There is no real ideal number of components – this dependent on the required functionality of the system – however, the fewer components the better, in terms of complexity.

The ideal design concept is ‘**Highly Cohesive but Loosely Coupled**’. This ideal result is for
CBS to be built from components which fit together with little glue code and with the ability to add/remove components (functionality) easily.
The same principle is followed for the production of glue code/integration code.
NOTE: the number of different COTS components used on the project are covered in the Software matrix document (in SE Teamroom)
SWIFT carries a lot of weight within the banking industry. The result is that customers of the SWIFT products have to follow their upgrade strategy – the size of the COTS component customer within the COTS vendor’s pond has a direct bearing on the vendor’s willingness to modify their products on the customer’s request.
The SWIFT 6 upgrade had numerous pre requisites – there was the requirement to upgrade AIX from r520 to r530. This in turn initiated the need to move from PSSP (which isn’t supported on r530) to CSM.
Therefore, a main TCO factor of CBSs is the number of prerequisite maintenance activities required, not just the number of components used to build the solution. For example, SAA also required DCE to be installed on AIX.
Because SWIFT’s standing within the banking arena they publish the upgrade roadmap for their products well in advance.
The maturity of components also appears to be an important cost factor. IE if a component has been around for a long time and is used by numerous clients there tends to be a better support structure for these products.
Therefore, best to use industry standard (best of breed) components in a solution. Websphere MQ is an industry standard for middleware.

Open coding: (Application Architect) interview open coding

<table>
<thead>
<tr>
<th>CodeNo.</th>
<th>Key Text</th>
<th>Early Open Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARiF1</td>
<td>Websphere is designed around a common architecture</td>
<td>Common architecture</td>
</tr>
<tr>
<td>ARiF2</td>
<td>each Websphere component has an MQ interface (MQ is used to send messages between different components and applications)</td>
<td>Aiding component integration, Reducing integration costs</td>
</tr>
<tr>
<td>ARiF3</td>
<td>COTS components supplied by different vendors are more of a factor affecting total system life costs than just the number of different components</td>
<td>Disparate vendors, Cost attributing factor</td>
</tr>
<tr>
<td>ARiF4</td>
<td>One main issue is the number of dependencies different vendor’s components have</td>
<td>Underlying dependencies</td>
</tr>
<tr>
<td>ARiF5</td>
<td>when upgrading one component other vendors’ components may also need to be upgraded</td>
<td>Managing component dependencies</td>
</tr>
<tr>
<td>ARiF6</td>
<td>Main problems with the maintenance of COTS systems, which has had big influences on the life costs:</td>
<td>Maintaining CBS influencing total life cost</td>
</tr>
<tr>
<td>ARiF7</td>
<td>Keeping each product in support when different products have different life spans</td>
<td>Managing multiple vendors components</td>
</tr>
<tr>
<td>ARiF8</td>
<td>Hidden dependencies - It was not always obvious that certain dependencies existed until the requirement to upgrade another product</td>
<td>Underlying dependencies</td>
</tr>
<tr>
<td>CodeNo.</td>
<td>Key Text</td>
<td>Early Open Codes</td>
</tr>
<tr>
<td>--------</td>
<td>----------</td>
<td>------------------</td>
</tr>
<tr>
<td>ARiF9</td>
<td>There is no real ideal number of components</td>
<td>Rejecting set number of components in design</td>
</tr>
<tr>
<td>ARiF10</td>
<td>this dependent on the required functionality of the system</td>
<td>Matching number of components to required functionality</td>
</tr>
<tr>
<td>ARiF11</td>
<td>however, the fewer components the better, in terms of complexity</td>
<td>Fewer components reducing system complexity</td>
</tr>
<tr>
<td>ARiF12</td>
<td>The ideal design concept is ‘Highly Cohesive but Loosely Coupled’</td>
<td>Designing for change</td>
</tr>
<tr>
<td>ARiF13</td>
<td>This ideal result is for COTS-based systems to be built from components which fit together with little glue code and with the ability to add/remove components (functionality) easily</td>
<td>Identifying design principle  Avoiding integration effort</td>
</tr>
<tr>
<td>ARiF14</td>
<td>The same principle is followed for the production of glue code/integration code.</td>
<td>Designing for change</td>
</tr>
<tr>
<td>ARiF15</td>
<td>SWIFT carries a lot of weight within the banking industry</td>
<td>Recognising industry sector leader</td>
</tr>
<tr>
<td>ARiF16</td>
<td>The result is that customers of the SWIFT products have to follow their upgrade strategy</td>
<td>Customers adhering to vendors upgrade schedule</td>
</tr>
<tr>
<td>ARiF17</td>
<td>the size of the COTS component customer within the COTS vendor’s area of concern</td>
<td>Customer status</td>
</tr>
<tr>
<td>ARiF17a</td>
<td>has a direct bearing on the vendor’s willingness to modify their products on the customer’s request</td>
<td>influencing vendor support quality</td>
</tr>
<tr>
<td>ARiF18</td>
<td>The SWIFT 6 upgrade had numerous pre requisites – there was the requirement to upgrade AIX from r520 to r530. This in turn initiated the need to move from PSSP (which isn’t supported on r530) to CSM</td>
<td>Managing dependencies</td>
</tr>
<tr>
<td>ARiF19</td>
<td>Therefore, a main TCO factor of CBSs is the number of prerequisite maintenance activities required, not just the number of components used to build the solution</td>
<td>Maintenance dependencies affecting cost</td>
</tr>
<tr>
<td>ARiF20</td>
<td>Because SWIFT’s standing within the banking arena they publish the upgrade roadmap for their products well in advance</td>
<td>Receiving upgrade schedule</td>
</tr>
<tr>
<td>ARiF21</td>
<td>The maturity of components also appears to be an important cost factor</td>
<td>Appreciating component maturity</td>
</tr>
<tr>
<td>ARiF22</td>
<td>if a component has been around for a long time and is used by numerous clients there tends to be a better support structure for these products.</td>
<td>Popularity defining component quality</td>
</tr>
<tr>
<td>ARiF23</td>
<td>best to use industry standard (best of breed) components in a solution. Websphere MQ is an industry standard for middleware</td>
<td>Best of breed</td>
</tr>
</tbody>
</table>
Concepts which may come in useful
Glutch & Weinstock (1997) state that to use COTS components effectively the system must be designed for change because the end user normally has no control over when the system will need to be changed. The main source of change is from vendors supplying upgrades to components. These tend to range from maintenance releases and minor upgrades to major upgrades and technology refreshes. The upgrade costs to the customer of a technology refresh tend to be higher than a minor upgrade due to the amount of effort involved. Concepts from this reference relating to designing for change are: layering, packaging, information hiding

Concept **Cost Reducing Strategy** and **design objective**
Comparing Common architecture ARiF1 with Aiding component integration ARiF2
Selecting components built around a common architecture is a Cost Reducing Strategy employed by system developers for aiding component integration because each component normally has a compatible interface requiring little modification.

The selection of components supporting a common architecture can also be seen to be a design objective of system architects who aim to select components with interfaces requiring minimal modification thus aiding component integration.

Concept **reducing integration effort**
Comparing Common architecture ARiF1 with Reducing integration costs ARiF2

The result of integrating components supporting a common architecture can be reducing integration costs for system developers because the components are normally supplied with common interfaces, thus leading to reducing integration effort.

Concept **Design Objective**
Comparing Common architecture ARiF1 with Disparate vendors ARiF3
A Design Objective of COTS system developers is choosing components supporting a common architecture, whether they are supplied by the same vendor or disparate vendors, because this can lessen the need to modify components, change interfaces or create glue code to enable the integration of the components into a system.

From the data (interview ARiF) there was a greater likelihood that components supplied by the same vendor would support the same standard or architecture. However, there is no guarantee that components supplied by the same vendor would necessarily support all system requirements. Therefore, conceptually, the CBD process should allow for other vendor’s products to be slotted into the system if they provided the added functionality. It can be seen that for this to work with minimal effort all of the components should support the same architectural standard.

Concept **increasing maintenance complexity**
Comparing Disparate vendors ARiF3 with Underlying dependencies ARiF4
The link between the 2 codes is concept of increasing maintenance complexity because with CBS built from components supplied by different vendors there can be different underlying dependencies of each vendor’s components adding to the complexity of maintaining the systems. For example, upgrading one vendor’s component may require other components to be upgraded in order for a system to continue functioning. Therefore, maintenance complexity is defined in terms of the number of factors which have to be considered during the life of a system. The number of different vendor’s products involved in a system can contribute to maintenance complexity.
**Concept: Degree of dependencies**
Comparing Underlying dependencies ARiF4 with Managing component dependencies ARiF5

The **Degree of dependencies** linking the number of underlying dependencies of different vendors components and the effort of managing component dependencies when maintaining systems can contribute to ongoing system maintenance costs for system administrators because the greater the degree of complexity i.e. more underlying dependencies between components can lead to greater time, effort and skill, to assess the effects of component dependencies and the effect of changing components on system stability, as well as additional system down-time resulting from additional change activity. For example, the requirement to upgrade one component may necessitate upgrading other COTS components and other system parts.

Therefore, component **Degree of dependencies** can be defined in terms of the number of interrelated dependencies between COTS components and system parts and the effect on the system when making changes to one or more system parts.

<table>
<thead>
<tr>
<th>Concept</th>
<th>increasing maintenance complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparing Underlying dependencies ARiF4 with Managing multiple vendors components ARiF7</td>
<td></td>
</tr>
</tbody>
</table>

The combination of codes Underlying dependencies ARiF4 and Managing multiple vendors’ components ARiF7 can contribute to **increasing maintenance complexity** for system maintainers because of 2 factors – When all vendors do not implement the same component upgrade cycle additional planning effort can be required by system maintainers to schedule all component upgrades. Secondly, with underlying dependencies between different vendors’ components, such as requiring other components to be at a specific release level or specific operating system versions, then the maintenance complexity can increase because many more factors have to be considered and other tasks performed, such as upgrading the operating system, when maintaining a system.

**Concept: increasing maintenance complexity**
Comparing Underlying dependencies ARiF4 with Appreciating hidden dependencies between components ARiF8

As above – with (Comparing Underlying dependencies ARiF4 with Managing multiple vendors components ARiF7)

**Concept: conflicting design decision, conflicting design principles and balancing design principles**
Comparing Underlying dependencies ARiF4 with Designing for change ARiF12

The ideal for COTS-based systems is Designing for change, involving using components which are highly cohesive, but loosely coupled. This facilitates a system which supports all system requirements whilst allowing components to be changed in isolation of the rest of the system. However, underlying dependencies between components can conflict with this ideal because it prevents changing a component in isolation of other system parts. Therefore, the link between the 2 codes can be the concept of a **conflicting design decision**.

The link between the 2 codes can also be **conflicting design principles** because
building CBS with underlying dependencies goes against the principle of designing for change.

However, the practicality of building CBS can result in system developers balancing design principles to include components with underlying dependencies because the available selection of loosely coupled components, which also satisfy the system requirements, may not exist, therefore, system developers can be forced to compromise on the design in order to produce a functioning system.

**Concept balancing design principles**  
Comparing Underlying dependencies ARiF4, ARiF8 with Fewer components reducing system complexity ARiF11

*Fewer components reducing system complexity* can be seen as a design decision made by COTS-based system developers aimed at reducing system complexity because of the reduced number of interconnections having to be created and managed between components. Conversely, selecting components with Underlying dependencies can lead to increasing maintenance complexity resulting from additional factors (knock-on requirement of upgrading further system parts when upgrading one component) having to be considered by system maintainers when maintaining COTS-based systems. Therefore, system designers are balancing design principles when selecting components with the aim of reducing system complexity, but using components (which may be the only available components supporting system requirements) with Underlying dependencies because reducing system complexity can contribute to reducing system costs, but using components with underlying dependencies can lead to increasing ongoing costs.
Appendix B7 - Interview with Project Manager (PMiG)

Contact summary form – field notes from interview:

<table>
<thead>
<tr>
<th>Contact Summary Form</th>
<th>Interview G (iG)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Contact name &amp; position:</strong></td>
<td></td>
</tr>
<tr>
<td>Benjamin Chance/Philadelphia/IBM@IBMUS</td>
<td>PM for TPA project</td>
</tr>
<tr>
<td><strong>Contact Type: Interview, Document etc.:</strong></td>
<td>Telephone interview</td>
</tr>
<tr>
<td><strong>Contact Date:</strong> 19:30 to 20:00</td>
<td></td>
</tr>
<tr>
<td><strong>Visit:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Phone:</strong></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES**

1. **Main issues and themes discussed:**

Factors affecting TCO of CBS

2. **Summary of information gathered (of failed to collect) on each target question:**

The most important factor was not the number of different COTS components but whether they were supplied by the same vendor or share a common API framework. Therefore, from a TCO perspective a higher number of components supplied by one vendor would cost less than managing less components supplied by multiple vendors.

3. **Other interesting points:**

4. **What further information required?**

---

**Interview with Benjamin Chance, PM for xxxx project**

In relation to the question on what factors affect the total cost (TCO) of ownership of a system built with (some) COTS components he stated that there is a perception within the IT Architect community that TCO will increase with the higher number of COTS components included within a system. This is because more components need to be connected together – it is the successful integration of components which tends to require more effort. With more components the continued maintenance of this integration requires more effort. For example, following the upgrade of a component the component may work differently. Thus, the stability of the system as a whole could be compromised. He suggested that the integration of any more than 5 components would result in much higher costs due to increased maintenance effort.

However, in practice the actual number of components included within a system isn’t the only TCO factor. For example, a higher number of COTS components which share a common API will cost much less to maintain than a lower number of components which don’t share this common API. Therefore, to minimise maintenance effort, and thus cost, the actual number of components is less significant. Furthermore, Integrating and maintaining a set of components supplied from different vendors is likely to require more effort than if the components were supplied from the same vendor and share the same common API framework.
For example, the integration of Mayberry Mail required the writing of a different API in order to successfully integrate it into the rest of the system.

A useful cost saving addition to component integration is if someone has already written an interface between 2 or more applications. For example. The integration of DB/2 into WebSphere was made easier because an interface already existed.

Open coding: (Project Manager) interview open coding

<table>
<thead>
<tr>
<th>CodeNo.</th>
<th>Key Text</th>
<th>Early Open Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMiG1</td>
<td>perception within the IT Architect community</td>
<td>IT Architect’s perception</td>
</tr>
<tr>
<td>PMiG2</td>
<td>TCO will increase with the higher number of COTS components included within a system</td>
<td>Increasing cost</td>
</tr>
<tr>
<td>PMiG3</td>
<td>more components need to be connected together</td>
<td>Connecting more components</td>
</tr>
<tr>
<td>PMiG4</td>
<td>successful integration of components which tends to require more effort</td>
<td>Increasing effort</td>
</tr>
<tr>
<td>PMiG5</td>
<td>With more components the continued maintenance of this integration requires more effort</td>
<td>Ongoing maintenance Increasing effort</td>
</tr>
<tr>
<td>PMiG6</td>
<td>following the upgrade of a component the component may work differently</td>
<td>Consequence of change</td>
</tr>
<tr>
<td>PMiG7</td>
<td>stability of the system as a whole could be compromised</td>
<td>Consequence of change</td>
</tr>
<tr>
<td>PMiG8</td>
<td>integration of any more than 5 components</td>
<td>Integration threshold</td>
</tr>
<tr>
<td>PMiG9</td>
<td>would result in much higher costs due to increased maintenance effort</td>
<td>Increasing system maintenance costs</td>
</tr>
<tr>
<td>PMiG10</td>
<td>In practice the actual number of components included within a system isn’t the only TCO factor</td>
<td>Ongoing cost complexity</td>
</tr>
<tr>
<td>PMiG11</td>
<td>a higher number of COTS components which share a common API will cost much less to maintain than a lower number of components which don’t share this common API</td>
<td>Increasing number of components Sharing common APIs Reducing costs</td>
</tr>
<tr>
<td>PMiG12</td>
<td>Integrating and maintaining a set of components supplied from different vendors is likely to require more effort than if the components were supplied from the same vendor and share the same common API framework</td>
<td>Vendor homogeneity Sharing common APIs</td>
</tr>
<tr>
<td>PMiG13</td>
<td>integration of Mayberry Mail required the writing of a different API</td>
<td>Integration programming</td>
</tr>
<tr>
<td>PMiG14</td>
<td>cost saving addition to component integration</td>
<td>Saving money</td>
</tr>
<tr>
<td>PMiG15</td>
<td>is if someone has already written an interface between 2 or more applications</td>
<td>Writing compatible interfaces</td>
</tr>
<tr>
<td>PMiG16</td>
<td>integration of DB/2 into WebSphere was made easier because an interface already</td>
<td>Exploiting existing interfaces</td>
</tr>
</tbody>
</table>
Initial memos: Memos (Ben Chance – PM - Interview)

Concept Increasing maintenance complexity, Increasing effort, Redoing integration work
Comparing codes IT Architect’s perception PMiG1, Increasing cost PMiG2, Connecting more components PMiG3, Increasing effort PMiG4, PMiG5, Ongoing maintenance PMiG5, Consequence of change PMiG6

From the data the perception of IT architects is that a consequence of including more components in COTS-based systems is increasing effort during two phases:

1) System development: The assumption is that Connecting more components PMiG3 can result in increasing effort because additional tasks needing to be performed when initially connecting interfaces of more components (From the interview there were no details on the action tasks required when integrating components, however, the assumption was that the modification required to enable two interfaces to work together requires programming effort to ensure compatibility of data being passed from one component to another – with more components the data format may have to be changed many times to facilitate data compatibility between components).

The relationship between codes when developing systems is:

The consequence of Connecting more components PMiG3 can be Increasing effort PMiG4 which contributes to Increasing cost PMiG2

2) System maintenance: Ongoing maintenance PMiG5 indicates that system maintenance is not a one-time activity but involves ongoing tasks performed throughout the life of the system. Maintenance activity normally involves change, such as upgrading components (In such cases vendors supply new versions of components to its customers. Installing upgraded components involves change to a system because the original component is replaced).

In addition to the effort required to effect change (Installing an upgraded component requires effort in planning, assessing effect and performing the change) increasing effort can be required when addressing the Consequence of change PMiG6. For example, a Consequence of change PMiG6 can be upgraded components working differently from original components (the nature of this difference varies – however, change can affect system stability). Therefore, addressing the consequence of change can effectively involve redoing integration work (such as, modifying, rewriting, testing configuration, wrapper or glue code), all requiring effort.

Therefore, if the Consequence of change PMiG6 to one component results in Increasing effort PMiG4 it can be assumed that the Consequence of change PMiG6 on systems comprising of increasing numbers of components is greater levels of increasing effort increase because of the increased number of individual connections between components to be managed (Increasing maintenance complexity). Potentially more configuration, wrapper or glue code may need to be modified,
rewritten and tested.

The relationship between codes when maintaining systems is:

The consequence of Connecting more components PMiG3 can lead to **Increasing maintenance complexity** resulting in Increasing effort PMiG4 needed to address the Consequence of change PMiG6 contributing to Increasing cost PMiG2

<table>
<thead>
<tr>
<th>Concept</th>
<th>Balancing design principles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparing codes Increasing number of components PMiG11, Sharing common APIs PMiG11 and Reducing costs PMiG11</td>
<td></td>
</tr>
</tbody>
</table>

System developers can be seen to be balancing design principles if considering Increasing number of components PMiG11, which can lead to increasing cost as a result of the effort required to connect and maintain additional connections between components., and the principle of selecting components Sharing common APIs PMiG11, which can lead to lower costs as a result of less effort required to integrate and maintain components supporting the same API.

The link between codes in Increasing number of components PMiG11 and Sharing common APIs PMiG11 indicates that in isolation the number of components used in a COTS-based system is not the main cost factor; the underlying architectural standard is also an important cost factor.

It should be stated that the ideal scenario is where less components are used share the same API or architectural standard

<table>
<thead>
<tr>
<th>Concept</th>
<th>Design Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparing codes Vendor homogeneity PMiG12, Sharing common APIs PMiG12</td>
<td></td>
</tr>
</tbody>
</table>

The link between these two codes is the concept of Design Decision. The concept of Vendor homogeneity implies that components purchased from the same vendor will have been built by the vendor around common APIs or common architectural standards, enabling integration and ongoing maintenance with less effort. Thus it can be a **design decision** to source components from the same vendor for this reason, rather than choosing components supplied from a variety of vendors.

The choice to use components which are supplied by the same vendor can be seen to be a **cost reducing strategy** if its occurrence was a conscious decision employed with the aim of reducing costs. If this is not the case sourcing components from the same vendor may occur as consequence of the absence of a better selection of components.
Appendix B8 - Interview with Project Manager (PMiH)

Contact summary form – field notes from interview:

<table>
<thead>
<tr>
<th>Contact Summary Form</th>
<th>Interview H (iH)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contact name &amp; position:</td>
<td>Project Manager</td>
</tr>
<tr>
<td>– project.</td>
<td></td>
</tr>
<tr>
<td>Contact Type: Interview, Document etc.:</td>
<td>Telephone interview</td>
</tr>
<tr>
<td>Contact Date:</td>
<td></td>
</tr>
<tr>
<td>Visit:</td>
<td></td>
</tr>
<tr>
<td>Phone:</td>
<td>(Tuson, AZ) 19:10 to 19:40</td>
</tr>
<tr>
<td>UK GMT</td>
<td></td>
</tr>
</tbody>
</table>

NOTES

1. Main issues and themes discussed:
   The cost factors affecting systems built from COTS components

2. Summary of information gathered (of failed to collect) on each target question:

3. Other interesting points:

4. What further information required?

Telephone Interview with , Project Manager

Question: What are the factors which affect the Total Cost of Ownership of a COTS-based system? How does the number of components used to build a CBS affect the TCO?

The Justify application was a tool which has been around for a long time. It was originally written in OS2. It has now been written in a Windows API. It is a reporting programme used by sales personnel to take US tax laws into consideration when producing reports. Currently it incorporates 1 COTS package.

It is used solely for internal use – It was considered to be useful for Business Partners. However, it was not possible to release the whole product to Business Partners (BPs) due to legal constraints on passing on a COTS product to additional parties.

The custom development part of the project worked well because there was a dedicated application support team in Tucson which developed and supported this (and other) products. Dallas could not give any more details on the cost factors of this project. However, he said that maintenance costs are reduced with as fewer interfaces as possible because there are fewer potential problems. Managing change is the main cause of ongoing costs because there can be many unknown factors.

Certain components have been designed to integrate together – eg Websphere Application Server, DB/2 and WebSphere MQ. This reduces the long term maintenance costs because when one component is updated by the vendor the other components are updated and their integration tested.

If the company’s business processes don’t fit the with the CBS approach this can cause a problem. It can be difficult to get a company to change its business processes. For some large organisations different processes are followed by different departments. For example, for an order status tool there was an attempt to consolidate the status of different product lines from different internal departments. However, there were requirements for specific programme logic for each product type. There was also a requirement to produce a common presentation.
method. However, there were so many special cases it was decided to go with a tailored application as it was deemed organisationally and economically impractical to get the internal departments to follow the same processes.

Open coding: (Project Manager) interview open coding

<table>
<thead>
<tr>
<th>CodeNo.</th>
<th>Key Text</th>
<th>Early Open Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMiH1</td>
<td>The <em>Justify</em> application was a tool which has been around for a long time</td>
<td>Legacy implementation</td>
</tr>
<tr>
<td>PMiH2</td>
<td>It was originally written in OS2. It has now been written in a Windows API.</td>
<td>Changing over time</td>
</tr>
<tr>
<td>PMiH3</td>
<td>Currently it incorporates 1 COTS package</td>
<td>Integrating COTS package</td>
</tr>
<tr>
<td>PMiH4</td>
<td>It was not possible to release the whole product to Business Partners (BPs) due to legal constraints on passing on a COTS product to additional parties</td>
<td>Legal constraints</td>
</tr>
<tr>
<td>PMiH5</td>
<td>The custom development part of the project worked well because there was a dedicated application support team</td>
<td>Custom development Providing dedicated support</td>
</tr>
<tr>
<td>PMiH6</td>
<td>Maintenance costs are reduced with as fewer interfaces as possible because there are few fewer potential problems</td>
<td>Reducing system maintenance costs Reducing potential problems</td>
</tr>
<tr>
<td>PMiH7</td>
<td>Managing change is the main cause of ongoing costs because there can be many unknown factors</td>
<td>Managing change Dealing with unknown factors</td>
</tr>
<tr>
<td>PMiH8</td>
<td>Certain components have been designed to integrate together eg Websphere Application Server, DB/2 and WebSphere MQ</td>
<td>Designing for integration</td>
</tr>
<tr>
<td>PMiH9</td>
<td>This reduces the long term maintenance costs because when one components is updated by the vendor the other components are updated and their integration tested</td>
<td>Reducing maintenance costs</td>
</tr>
<tr>
<td>PMiH10</td>
<td>If the company’s business processes don’t fit with the CBS approach this can cause a problem</td>
<td>Problems relating business processes to COTS-based approach</td>
</tr>
<tr>
<td>PMiH11</td>
<td>It can be difficult to get a company to change its business processes</td>
<td>Resisting business process change</td>
</tr>
<tr>
<td>PMiH12</td>
<td>In some large organisations different processes are followed by different departments</td>
<td>Business process complexity</td>
</tr>
<tr>
<td>PMiH13</td>
<td>For example, for an order status tool there was an attempt to consolidate the status of different product lines from different internal departments</td>
<td>Attempting to consolidate business processes</td>
</tr>
<tr>
<td>PMiH14</td>
<td>However, there were requirements for specific programme logic for each product type.</td>
<td>Business requirement issues</td>
</tr>
<tr>
<td>PMiH15</td>
<td>There was also a requirement to produce a common presentation method.</td>
<td>Specific requirements</td>
</tr>
<tr>
<td>PMiH16</td>
<td>However, there were so many special cases</td>
<td>Resisting business process</td>
</tr>
</tbody>
</table>
it was decided to go with a tailored application as it was deemed organisationally and economically impractical to get the internal departments to follow the same processes.

Initial memos: Memo (Dallas Johnson – PM - Interview)

Concept managing change and Designing for change
Comparing Changing over time PMiH2 and Legacy implementation PMiH1

The link between these 2 codes shows that a consequence of time is change. Software systems can change over time – operating systems are upgraded, COTS-components evolve over time or are withdrawn and other changes can occur. Therefore, the concept of managing change is performed by system maintainers in order to keep systems running.

Managing change incurs cost for system maintainers because it involves effort and in some cases, specialist skills.

Designing for change can assist in managing change because it involves designing for parts of COTS-based systems to be changed with minimal impact the rest of the system. However, the design principle of Designing for change normally has to be thought of early on in the design process because it may not be possible to retrofit this design principle after system implementation.

Thus, with reference to Glaser’s (1978) theoretical codes a perceived consequence of implementing a designing for change policy is to make managing change easier (costing less, less onerous) for system maintainers because one or more parts of a system can be changed with little effect on the rest of the system.

Without designing for change a change to one system part may result in the knock-on effect of needing change other system parts.

Concept cost reducing strategy
Comparing Reducing system maintenance costs PMiH6 and Reducing potential problems PMiH6

Building systems with fewer components can be seen cost reducing strategy, if consciously employed by system developers, because it is perceived to result in reducing system maintenance costs PMiH6 byReducing potential problems PMiH6 caused by connecting fewer interfaces.

Concept Managing change and Increasing costs
Comparing Managing change PMiH7 and Dealing with unknown factors PMiH7

Change and the act of Managing change PMiH7 in COTS-based systems is a contributing factor of Increasing costs because of the effect of dealing with unknown factors, which may not be predicted when a system was originally conceived. For example, when a vendor releases an upgraded component requiring an upgraded version of the operating system.

Concept Resisting change
Comparing Resisting business process change PMiH11, PMiH16 and Attempting to
consolidate business processes PMiH13

From the data in interview (PMiH) **Resisting Change** by organisations can be a challenge the use of the COTS-based approach. Generally, organisations have business processes. The COTS-based approach can result in a need for business processes to change in order to match the way the COTS components work. There can be resistance to this change.

Furthermore, in some organisations different internal departments have different business processes. The use of the COTS-based approach can require a company to standardise processes across departments, which can lead to personnel resisting change. Producing custom processes for different departments can be a solution, but this can be difficult and add cost as due to the effort required to tailor COTS components.
**Appendix B9 - Interview with Project Manager (PMiJ)**

Contact summary form – field notes from interview:

<table>
<thead>
<tr>
<th>Contact Summary Form</th>
<th>Interview J (iJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Contact name &amp; position:</strong></td>
<td>Project Manager – project.</td>
</tr>
<tr>
<td><strong>Contact Type: Interview, Document etc.:</strong></td>
<td>Telephone interview</td>
</tr>
<tr>
<td><strong>Contact Date:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Visit:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Phone</strong></td>
<td>1-919-846-1676 (Raleigh) 19:15 to 19:50 UK GMT</td>
</tr>
</tbody>
</table>

**NOTES**

1. **Main issues and themes discussed:**
   - The cost factors affecting systems built from COTS components

2. **Summary of information gathered (of failed to collect) on each target question:**

3. **Other interesting points:**

4. **What further information required?**

---

**Telephone Interview with Ray Hart, Project Manager**

Question: What are the factors which affect the Total Cost of Ownership of a COTS-based system? How does the number of components used to build a CBS affect the TCO?

With regard to the Functional density rule of thumb the interviewee said that this doesn’t just apply to COTS-based systems – it can apply when attempting to integrate different chunks to custom code.

The major factor which can drive up maintenance costs is the requirement to keep track of all the dependent version numbers. He has known an application which was running on a version of ODBC which was 4 release levels behind the most recent. This worked successfully with a certain level of DB/2. However, there was a requirement to integrate an Oracle database – this wouldn’t work with the installed ODBC level – upgrading ODBC affected the DB/2 functionality.

There are other examples of hidden dependencies of various parts of systems. The upgrade of one part can adversely affect the functionality/stability of the system – therefore, much effort can be spent in restoring system stability.

Another example to the extension of system functionality causing problems was when attempting to integrate a product called BRIO into the PCCO Entitlement Warehouse system – the installed version of DB/2 was not compatible. Upgrading DB/2 stopped SQL working.

I asked xxx if it was better to build a system from a larger number of COTS components which were built to the same architectural framework (and perhaps supplied by the same vendor) than a lesser number of components which were supplied by different vendors and which did not adhere to the same architectural framework. He replied with “ah, you mean standards”. Sourcing a set of components which have supposedly been built to the same standard should introduce greater flexibility to the system. However, in his experience, after selecting components which the vendors purported to be built to a certain standard he’s found that they don’t in fact follow the published standard.
On function points Ray said that they are not always a good measure of system size. Duplicate inputs and outputs are not always counted.

<table>
<thead>
<tr>
<th>CodeNo.</th>
<th>Key Text</th>
<th>Early Open Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMiJ1</td>
<td>With regard to the Functional density rule of thumb xxxx said that this doesn’t just apply to COTS-based systems – it can apply when attempting to integrate different chunks to custom code</td>
<td>Functional Density Rule of Thumb affecting all code</td>
</tr>
<tr>
<td>PMiJ2</td>
<td>The major factor which can drive up maintenance costs is the requirement to keep track of all the dependent version numbers.</td>
<td>Tracking dependencies</td>
</tr>
<tr>
<td>PMiJ3</td>
<td>He has known an application which was running on a version of ODBC which was 4 release levels behind the most recent.</td>
<td>Tracking dependencies</td>
</tr>
<tr>
<td>PMiJ4</td>
<td>This worked successfully with a certain level of DB/2.</td>
<td></td>
</tr>
<tr>
<td>PMiJ5</td>
<td>However, there was a requirement to integrate an Oracle database – this wouldn’t work with the installed ODBC level – upgrading ODBC affected the DB/2 functionality.</td>
<td>Managing dependencies</td>
</tr>
<tr>
<td>PMiJ6</td>
<td>There are other examples of hidden dependencies of various parts of systems.</td>
<td>Hidden dependencies</td>
</tr>
<tr>
<td>PMiJ7</td>
<td>The upgrade of one part can adversely affect the functionality/stability of the system – therefore, much effort can be spent in restoring system stability.</td>
<td>Managing dependencies</td>
</tr>
<tr>
<td>PMiJ8</td>
<td>Another example to the extension of system functionality causing problems was when attempting to integrate a product called BRIO into the PCCO Entitlement Warehouse system – the installed version of DB/2 was not compatible. Upgrading DB/2 stopped SQL working.</td>
<td>Managing dependencies</td>
</tr>
<tr>
<td>PMiJ9</td>
<td>Sourcing a set of components which have supposedly been built to the same standard should introduce greater flexibility to the system.</td>
<td>Design principle</td>
</tr>
<tr>
<td></td>
<td>However, in his experience, after selecting components which the vendors purported to be built to a certain standard he’s found that they don’t in fact follow the published standard.</td>
<td>Conflicting Design principle</td>
</tr>
</tbody>
</table>

Initial memos: Memo (Ray Hart – PM - Interview)

**Concept** *Degree of dependency*
Comparing Tracking dependencies PMiJ2, PMiJ3 and Underlying dependencies PMiJ4
The link between codes Tracking dependencies PMiJ2, PMiJ3 and Underlying dependencies PMiJ4 is the concept of **Degree of dependency**.

Understanding the **Degree of dependency** between components and other system parts has implications on managing the cost of change. The effort required in Tracking dependencies PMiJ2, PMiJ3 and dealing with Underlying dependencies PMiJ4 when managing change can contribute to increasing costs because changing one system part can require other system parts to be changed if dependencies exist. Furthermore, without an understanding of the **Degree of dependency** between items the time when it becomes clear that other items need to be changed is when the change to one item causes the system to stop working – thus, the original estimated effort and cost to change one item could be exceeded.

**Concept Degree of dependency and managing change**
Comparing Tracking dependencies PMiJ2, PMiJ3, Underlying dependencies PMiJ4, Managing dependencies PMiJ5, PMiJ7, PMiJ8 and Hidden dependencies PMiJ6

The significance of tasks involved with Managing dependencies PMiJ5, PMiJ7, PMiJ8, which involve Tracking dependencies PMiJ2, PMiJ3 and dealing with the consequences of Underlying dependencies PMiJ4 and Hidden dependencies PMiJ6 can become significant when managing change because it is following change when the effects of dependencies become apparent. Without the requirement for change the degree of dependency between system parts would not matter one the system is functioning correctly. It is the act of change which can affect the equilibrium of a working system because as seen in the date (other interviews) changing one item can have the knock-on effect of requiring other system parts to also be changed.

Therefore, the **degree of dependency** between system parts can cause the act of **managing change** to contribute to increasing cost.

Conversely, the assumption that design principle **designing for change**, which includes reducing the dependency between system parts can contribute to reducing the effort required to perform change activity. However, knowledge of all future dependencies may not be able to be predicted during the system design phase because of the volatility of the COTS-component marketplace.

**Concept Lacking common understanding**
Comparing Design principle PMiJ9 and Failing to follow standard PMiJ10

Sourcing components supporting the same **standard** is considered to be a design principle because the assumption is that these components will integrate (with other components and system parts) with minimal effort and result in less effort when maintained.

However, from the data (different interviews) details on what constitutes the same standard varied. In some instances the same standard was taken to mean ‘supporting the same API’. In other instances it was taken to mean supporting the same architectural standard. However, for the purpose of this study supporting a common standard is taken to mean that there is a similarity in the components which facilitates their integration and maintenance with minimal effort. Conversely, the integration of components supporting different standards normally requires additional effort, and this cost, in order to manipulate and standardise the communication between components and other system parts.
Codes Design principle PMiJ9, when vendors purport components to adhere to a given standard, can link to the concept Lacking common understanding when the components are found not to support the published standard (code Failing to follow standard PMiJ10). Lacking common understanding relates to the details publicised by vendors and the expectations of their customers when these expectations are not met.

However, from the data no reasons were given for this. It may be surmised that a vendor’s interpretation of the standard may not be the same as that of their customers. There may be cultural differences with the interpretation of the language used by vendors to describe their product’s functionality by customers in other parts of the world. This is an area for further investigation as the consequence selecting components which prove not to function as described can be costly in time and effort. Time and effort can be wasted testing an inappropriate component. Additional time and effort could be spent in locating and testing alternative components or in modifying (if possible) and testing the original component.

Therefore, the following relationship can exist:

Lacking common understanding can contribute to → Increasing cost
Appendix B10 - Interview with Infrastructure Architect (ARiK)

Contact summary form – field notes from interview:

<table>
<thead>
<tr>
<th>Contact Summary Form</th>
<th>Interview K (iK)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Contact name &amp; position:</strong></td>
<td>Stuart Wilson - <a href="mailto:stuart_wilson@uk.ibm.com">stuart_wilson@uk.ibm.com</a> Infrastructure Architect</td>
</tr>
<tr>
<td><strong>Contact Type: Interview, Document etc.:</strong></td>
<td>Face to Face Interview</td>
</tr>
<tr>
<td><strong>Contact Date:</strong></td>
<td>Visit: North Harbour café 15:00 to 16:00</td>
</tr>
<tr>
<td><strong>Phone:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>NOTES</strong></td>
<td></td>
</tr>
<tr>
<td><strong>1. Main issues and themes discussed:</strong></td>
<td>Factors which affect the Total Cost of Ownership of a COTS-based system</td>
</tr>
<tr>
<td><strong>2. Summary of information gathered (of failed to collect) on each target question:</strong></td>
<td>With regard to TCO the number of COTS components supplied from diverse vendors had the greatest effect</td>
</tr>
<tr>
<td><strong>3. Other interesting points:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>4. What further information required?</strong></td>
<td></td>
</tr>
</tbody>
</table>

Interview with Stuart Wilson, Infrastructure Architect

Question: What are the factors which affect the Total Cost of Ownership of a COTS-based system? How does the number of components used to build a CBS affect the TCO?

COTS in xxx project seen to be cheaper than creating bespoke software.
PayPlus
Problems which have been encountered which have a bearing on the TCO:

a) Functionality: if the component/product offers too much functionality. The integrator needed to spend time understanding what functionality each component could offer before isolating and understanding the scope of the required functionality.

b) Secondary skills were required. For example, in order to implement the PayPlus product additional skills in SQL, Host Integration Server and Websphere MQ were required.

c) Code quality. With Payplus the vendor, Fundtech, produced a low quality of code. Each release of this product tended to contain many software bugs. They have also forced the xxx project team to upgrade PayPlus every year.

d) Number of pre-requisites: Payplus is a product which has many pre-requisite software components. These all need to be kept in line in order to avoid compatibility issues.

e) Infrastructure requirements: Functionally Payplus needed to function correctly. However, it also needed to work with the total solution ie be Highly Available. Additional effort was required to design and implement a solution to facilitate the replication of the PayPlus database across different sites WR1121 was the project to upgrade the Replicator.
With regard to TCO the number of COTS components supplied from diverse vendors had the greatest effect. Getting timely support from the vendor was one major problem. However, the need to discuss problems with the vendors has been rare. If possible, sourcing components from the one vendor is better - ie one normally has to deal with the one support team.

Integration issues are the customers’ problem as when PayPlus was originally sourced Fundtech provided a ‘certification’ that the product would work in the planned environment ie W2K. However, in practice PayPlus was installed on a W2K build with a different fixpack from the original certification declaration.

System engineering (SE) classify the size of a system/project as S.M or L. The total cost breakdown was 20% design, 80% build, test and implementation

The Payplus project involved integration of 4 main components:
- Host integration server and SQL Server (MS products)
- MQ Websphere (IBM)
- PayPlus (Funtech)
- It has cost 300 man days per year of effort to maintain this set of products.
- Payplus was selected due to regularity issues ie the FED will only sanction certain certified products. PayPlus was one of these products.

Nokia replacement project: The fact that the Nokia Network sensors were going out of support and there was a massive investment in the ISS SiteProtector IDS system it was decided that the G400 network sensors would be a suitable replacement because they were supported by ISS. However, there was a lack of understanding of the G400 product which led to high implementation and maintenance costs.

With the Level 7 Firewalls: additional costs (implementation costs) were as a result of the implementation team discovering that functionally they worked differently to the way first envisaged.

Athene was rubbish. It didn’t work for performance reporting. The problem with some bespoke products is when the developers leave there is no one to maintain it. Other issues with bespoke systems are that business requirements Change over time. The customer would have implemented a COTS solution for SI if it existed (this would have led to a faster time to market).

However, COTS vendors are reluctant to add new features to there products. If they do it’s normally expensive.

The main cost factors are:
The number of COTS products + the number of different vendors + the regulatory factors

Open coding: (Infrastructure Architect) interview open coding

<table>
<thead>
<tr>
<th>CodeNo.</th>
<th>Key Text</th>
<th>Early Open Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARiK1</td>
<td>COTS in xxx project seen to be cheaper than creating bespoke software</td>
<td>Perception of cost saving</td>
</tr>
<tr>
<td>ARiK2</td>
<td>Problems which have been encountered which have a bearing on the TCO:</td>
<td>Cost drivers</td>
</tr>
<tr>
<td>ARiK3</td>
<td>a) Functionality: if the component/product offers too much functionality</td>
<td>Providing too much functionality</td>
</tr>
<tr>
<td>ARiK4</td>
<td>The integrator needed to spend time understanding what functionality each component could offer before isolating and understanding the scope of the required</td>
<td>Assessing component functionality</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Disabling functionality</td>
</tr>
<tr>
<td>CodeNo.</td>
<td>Key Text</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>ARiK5</td>
<td>b) Secondary skills were required. For example, in order to implement the PayPlus product additional skills in SQL, Host Integration Server and Websphere MQ were required.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Requiring specialist skill</td>
<td></td>
</tr>
<tr>
<td>ARiK6</td>
<td>c) Code quality. With Payplus the vendor, Fundtech, produced a low quality of code. Each release of this product tended to contain many software bugs. They have also forced the xxx project team to upgrade PayPlus every year.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Poor product quality</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vendor forcing upgrade</td>
<td></td>
</tr>
<tr>
<td>ARiK7</td>
<td>d) Number of pre-requisites: Payplus is a product which has many pre-requisite software components. These all need to be kept in line in order to avoid compatibility issues.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Managing dependencies</td>
<td></td>
</tr>
<tr>
<td>ARiK8</td>
<td>e) Infrastructure requirements: Functionally Payplus needed to function correctly. However, it also needed to work with the total solution ie be Highly Available.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Providing suitable infrastructure</td>
<td></td>
</tr>
<tr>
<td>ARiK9</td>
<td>With regard to TCO the number of COTS components supplied from diverse vendors had the greatest effect.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vendor homogeneity</td>
<td></td>
</tr>
<tr>
<td>ARiK10</td>
<td>Getting timely support from the vendor was one major problem. However, the need to discuss problems with the vendors has been rare.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Quality of support</td>
<td></td>
</tr>
<tr>
<td>ARiK11</td>
<td>If possible, sourcing components from the one vendor is better - ie one normally has to deal with the one support team</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sourcing components from one vendor</td>
<td></td>
</tr>
<tr>
<td>ARiK12</td>
<td>Integration issues are the customers’ problem as when PayPlus was originally sourced Fundtech provided a ‘certification’ that the product would work in the planned environment ie W2K. However, in practice PayPlus was installed on a W2K build with a different fixpack from the original certification declaration.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vendor certifying component</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Certification limitation</td>
<td></td>
</tr>
<tr>
<td>ARiK13</td>
<td>Nokia replacement project: The fact that the Nokia Network sensors were going out of support and there was a massive investment in the ISS SiteProtector IDS system it was decided that the G400 network sensors would be a suitable replacement because they were supported by ISS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Changing technologies</td>
<td></td>
</tr>
<tr>
<td>ARiK14</td>
<td>However, there was a lack of understanding of the G400 product which led to high implementation and maintenance costs.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Lacking product understanding</td>
<td></td>
</tr>
<tr>
<td>ARiK15</td>
<td>With the Level 7 Firewalls: additional costs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Lacking product</td>
<td></td>
</tr>
<tr>
<td>CodeNo.</td>
<td>Key Text</td>
<td>Early Open Codes</td>
</tr>
<tr>
<td>--------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td></td>
<td>(implementation costs) were as a result of the implementation team discovering that functionally they worked differently to the way first envisaged.</td>
<td>understanding</td>
</tr>
<tr>
<td>ARiK16</td>
<td>Athene was rubbish. It didn’t work for performance reporting. The problem with some bespoke products is when the developers leave there is no one to maintain it. Other issues with bespoke systems are that business requirements Change over time.</td>
<td>Losing skill, Changing business requirements</td>
</tr>
<tr>
<td>ARiK17</td>
<td>However, COTS vendors are reluctant to add new features to their products. If they do it’s normally expensive.</td>
<td></td>
</tr>
<tr>
<td>ARiK18</td>
<td>The main cost factors are: The number of COTS products + the number of different vendors + the regulatory factors</td>
<td></td>
</tr>
</tbody>
</table>

Initial memos: Memo (Stuart Wilson – Infrastructure Architect - Interview)

Concept **cost reducing strategy**
Comparing Perception of cost saving ARiK1

The use of COTS-components within IBM is perceived to cost less than producing custom software. Therefore, with reference to Glaser’s (1978) theoretical coding family if the use of COTS-components are consciously chosen for the purpose of saving cost then the use of COTS components can be seen to be a **cost reducing strategy**.

Concept **design decision** and **balancing cost challenges**
Providing too much functionality ARiK3, Assessing component functionality ARiK4 and Disabling functionality ARiK4

*Providing too much functionality ARiK3* can be an issue with COTS components. A **design decision** may be to select specific components to provide functionality to support the system requirements. However, the additional effort required in Assessing component functionality ARiK4 and possibly *Disabling functionality ARiK4* can contribute to increasing costs because the cost saving of using COTS components can be outweighed by the additional effort of assessing and disabling additional functionality, tasks which also tend to require specific skills. Therefore, system developers are **balancing cost challenges** of using COTS components when performing these tasks ie balancing the challenge of reducing costs by using COTS with the added costs of making modifications to disable functionality

Concept **increasing cost**
Requiring specialist skill ARiK5

Integration tasks (and maintenance tasks) *Requiring specialist skill ARiK5* can contribute to **increasing cost** because the cost of hiring people with these skills is normally higher than the costs required to hire normal support staff.

With reference to Glaser’s (1978) theoretical codes the relationship between
Requiring specialist skill ARiK5 and increasing cost can be explained in terms of degree. There is a cost associated with the hiring of all people. The issue here is a ‘Question of degree’ because the skills required to perform component integration and maintenance tasks tend to be more specialised and thus, incur higher cost.

Concept Beyond sphere of influence and conflicting business motives
Poor product quality ARiK6 and Vendor forcing upgrade ARiK6

The effect of Poor product quality ARiK6 of COTS components and Vendor forcing upgrade ARiK6 of components is often be Beyond the sphere of influence of system developers and maintainers (the vendor’s customers) because it is vendors who are responsible for the quality and maintenance of their products, not the vendor’s clients who have purchased the components.

The problem is that there can be conflicting business motives between vendor and customer. The aim of the vendor is to make a profit from selling identical copies of a component, providing fixes to groups of bugs in a patch or upgrade which is released at specific times. This process may help to keep their cost down as they can plan for the fixes (or improved features) they need to produce between patches or upgrades. However, the motive of the customer is to purchase a product, which functions correctly, with which to incorporate within a system, which in turn is (normally) used to make a profit.

With reference to Glaser’s (1978) ‘degree’ theoretical coding family there may be different degrees of influence between vendor and customer to get software bugs fixed or to modify the product upgrade cycle. However, this appears to be dependent upon several factors, such as, the status of the vendor within the COTS marketplace, the status of the customer in the eyes of the vendor or the flexibility and determination of the vendor’s organisation to respond to individual customer requests. For example, some vendors, who are considered to be industry leaders within their field of products, may be less willing to fix specific software problems for individual customers in addition to bundling up a group of fixes within a patch, which may take time to materialise. Conversely, vendors who are keen to supply components to specific customers may be willing to be more flexible.

Concept increasing cost and degree of dependencies and managing change
Managing dependencies ARiK7

The effort required in Managing dependencies ARiK7 between components is dependent upon the degree of dependencies between components and contributes to increasing cost in two ways:

During system development: The dependencies between all products and system parts have to be understood and compatible product levels installed in order to initially implement a COTS-based system. A consequence of a higher degree of dependency between system parts can be additional effort to plan and to test that the system works as a whole.

A result of change: There is a link between degree of dependencies, managing change and managing dependencies. Thus, the higher the degree of dependency, more effort can be spent on managing dependencies when managing change because it is change which is most likely to disrupt existing dependencies.
Regarding *dependencies*, it can be suggested that once all dependencies between components have been addressed and the system is functioning as designed *managing dependencies* would no longer need to occur if nothing *changed* with the system. It is *change* which is likely to cause the greatest disruption to dependencies (An example was given in interview ARiC where the degree of dependencies of systems 'sat in the corner running for years' supporting business processes which will not change, would have limited effect. This is assuming that the hardware keeps functioning and any licensing charges are paid).

However, the requirement for constant change is usual for most systems.

<table>
<thead>
<tr>
<th>Concept</th>
<th>degree of dependencies and managing change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Providing suitable infrastructure ARiK8</td>
<td></td>
</tr>
</tbody>
</table>

*Providing suitable infrastructure ARiK8* to enable COTS components to function correctly can be viewed as a dependency because the dependency for some components is a requirement of specific hardware and network infrastructure solutions. Therefore, the dependencies related to *Providing suitable infrastructure ARiK8* can be defined by *degree of dependencies* in that more dependencies would result in more issues to consider during system development and when *managing change*.

<table>
<thead>
<tr>
<th>Concept</th>
<th>design decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vendor homogeneity ARiK9</td>
<td></td>
</tr>
</tbody>
</table>

*Vendor homogeneity ARiK9* can be explained in terms of the degree of vendor homogeneity. From the data it is seen that the higher number of vendors supplying components to be included in a system the greater the effect will be on the total lifecycle cost of a system. This is because of the following:

More vendor/customer relationships have to be managed, requiring more effort when integrating and maintaining a system.

Each vendor may have different upgrade and patching schedules, thus managing the maintenance of components supplied from different vendors may require more effort.

At the other end of the continuum the assumption is that components supplied by the same vendor are likely to require less integration effort integrate than when integrating components supplied by different vendors.

Therefore, the concept of vendor homogeneity can be design decision employed by system developers.

<table>
<thead>
<tr>
<th>Concept</th>
<th>support quality and support complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vendor homogeneity ARiK9, Quality of support ARiK10 and Sourcing components from one vendor ARiK11</td>
<td></td>
</tr>
</tbody>
</table>

*Codes* *Vendor homogeneity ARiK9, Quality of support ARiK10 and Sourcing components from one vendor ARiK11* can be seen to be linked to the concepts of *support quality* and *support complexity*. *Vendor homogeneity ARiK* – the number of vendors contributing to supplying components to a system can be seen to have a bearing on the functional compatibility of components. However, from system developers’ perspective, the general assumption is that *vendor homogeneity* affects
the support complexity because as the number of vendors supplying components increases more support teams have to be contacted when dealing with problems, product upgrades, etc. Thus, support complexity can be defined in terms of number of different support teams.

The support quality of individual support teams may be high but system developer’s perception of having to deal with increasing numbers of support teams, as the number of components being supplied by different vendors increases, can collectively add to feeling of diminishing support quality (a single point of contact for gaining support for products supplied from multiple vendors could enhance support quality).

With reference to Glaser’s (1978) theoretical coding families Support quality and support complexity can be defined in terms of ‘degree’, measured by the size of vendor’s organisation. For example, the assumption from system developers is that acquiring components from one vendor customers are more likely to deal with one support team. At the ‘smaller vendor size’ end of the continuum this may be true as one team may support all products. In this case support quality tends to be higher as one support team can provide assistance for all products (however, it should be stated that a single support team can still provide poor quality service). Support complexity can be seen to be low as a customer has only to contact one support team for all product enquiries.

However, at the other end of the ‘vendor size’ continuum, as vendor organisations get larger each product may be supported by different support teams or departments within the same organisation. In such cases support quality can decrease because each support team/department can appear as separate organisations to the customer, with little synergy between each team/department (see Interview SDiE xxxxxx). Support complexity may increase as a result of customers having to contact more than one support team/department to receive support for different components supplied by the same vendor.
Appendix B11 - Interview with Infrastructure and Integration Architect (ARiL)

Contact summary form – field notes from interview:

<table>
<thead>
<tr>
<th>Contact Summary Form</th>
<th>Interview L (iL)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Contact name &amp; position:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Contact Type:</strong> Interview, Document etc.:</td>
<td>Face to Face Interview</td>
</tr>
<tr>
<td><strong>Contact Date:</strong></td>
<td>15:15 – 15:45</td>
</tr>
<tr>
<td><strong>Visit:</strong></td>
<td>North Harbour D1 Canteen</td>
</tr>
<tr>
<td><strong>Phone:</strong></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES**

1. Main issues and themes discussed:
   Factors which affect the Total Cost of Ownership of a COTS-based system

2. Summary of information gathered (of failed to collect) on each target question:
   Vendors Statement of Compatibility and good internal support structures important for reducing TCO.

3. Other interesting points:

4. What further information required?

---

Interview with Chris Latham, Infrastructure and Integration Architect

Question: What are the factors which affect the Total Cost of Ownership of a COTS-based system? How does the number of components used to build a CBS affect the TCO? What factors affect the economic life of a CBS?

Background comments:
Bespoke applications require a team of application support personnel to look after them.
On a previous project – xxxxx:
Many COTS components had an overlap of functionality. His role was to investigate which components could be removed without compromising the functionality of the system. For example, Vital (a capacity and performance COTS component) and a component supplied by Cisco both had an overlap of functionality. He found that each component had implicit dependencies – therefore the removal of one component may have resulted in unpredictable results. This piece of work was not completed as he was pulled off this work stream to perform other tasks.

With the xxxx project. It was clear that with an increase in the number of components integrated there is a requirement for more staff members in order to understand their functionality and understand them. Staffing is a major cost.

The greater number of components integrated the greater the complexity of a system because components have to communicate with other components and custom code. One solution is to use a ‘communication bus’ which serves to interpret and standardise the communication between different components. He has never implemented such a solution though.

Continuous patching of components requires effort. Therefore, the more components used the
more patching effort (and testing) required. However, with certain trusted COTS products, such as Norton Antivirus, there is no need to spend time testing the functionality of the product. The only testing is whether the product’s integration works. NOTE: there is a difference in the effort required to install and maintain Antivirus on a stand alone system compared with installing within an enterprise setting. There is the question of how to manage the continuous update of virus definitions across numerous systems. The same concept was used with PcAnywhere – integration testing was performed, not testing of the product.

Other long term cost factors of CBS: The scale of the COTS products used. For example, smaller COTS products, such as Norton AV is much easier (cheaper) to manage than much larger components, such as EMC’s Network Management product. With the latter the vendor struggled with providing support on the integration of this product within the computing environment. The number of different systems on which the solution has to be integrated on affects the maintenance costs.

Another project was involved within the ‘system maintenance’ lifecycle stage was considered to involve a change in functionality. For example, one banking system involving 2000+ ATMs with connections to many banks the main product was available from SDM. However, unlike traditional ‘black box’ components the source code was available for modification. This allowed the project team to modify the program to add ‘Chip and Pin’ functionality. Another similar banking solution involving a COTS solution where no source code access was available resulted in the customer requiring the vendor to modify their product to include Chip and Pin functionality. This process took a lot of time. Implementation of the EMC product became a challenge because the project did not have a dedicated Project Manager assigned to the project. Much integration work was required – the vendor did not supply a lot of support.

It is important to get the vendor involved early – vendor support is very important in 1) getting the product to work in the first place. 2) Keeping the product working over time. The main variables with affect the TCO of a CBS are:

The complexity of the components used.
The complexity of the system as a whole.
The number of bugs in both the COTS components and any bespoke code. considers the following to be classed as maintenance activities, and thus contribute to maintenance costs:

1. Patch management – to fix bugs or security vulnerabilities.
2. System enhancements.
4. Dependencies – For example, following the requirement to upgrade the OS the application may need to be recompiled.

Other general system cost factors:
A propriety system is better if the system is to be changed a lot over time – however, the support costs may be higher as may need a team of people on call to deal with any problems. When building a system from scratch Industry knowledge by the project team is important. For example, with the ATM solution it was important to understand the detail sent from a card chip. If using a COTS solution it may possible to just buy a product which is already built to the banking applicable standards.

Open coding: (Integration Architect) interview open coding

<table>
<thead>
<tr>
<th>CodeNo.</th>
<th>Key Text</th>
<th>Early Open Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARiL1</td>
<td>Many COTS components had an overlap of</td>
<td>Overlapping functionality</td>
</tr>
<tr>
<td>CodeNo.</td>
<td>Key Text</td>
<td>Early Open Codes</td>
</tr>
<tr>
<td>--------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>ARiL2</td>
<td>His role was to investigate which components could be removed without compromising the functionality of the system.</td>
<td>Investigating dependencies</td>
</tr>
<tr>
<td>ARiL3</td>
<td>He found that each component had implicit dependencies – therefore the removal of one component may have resulted in unpredictable results.</td>
<td>Underlying dependencies</td>
</tr>
<tr>
<td>ARiL4</td>
<td>This piece of work was not completed as he was pulled off this work stream to perform other tasks.</td>
<td>Re-prioritising tasks</td>
</tr>
<tr>
<td>ARiL5</td>
<td>with an increase in the number of components integrated there is a requirement for more staff members in order to understand their functionality and understand them.</td>
<td>Increasing numbers of components</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Requiring more staff</td>
</tr>
<tr>
<td>ARiL6</td>
<td>Staffing is a major cost</td>
<td>Staffing cost</td>
</tr>
<tr>
<td>ARiL7</td>
<td>The greater number of components integrated the greater the complexity of a system because components have to communicate with other components and custom code.</td>
<td>Increasing system complexity</td>
</tr>
<tr>
<td>ARiL8</td>
<td>One solution is to use a ‘communication bus’ which serves to interpret and standardise the communication between different components.</td>
<td>Communication bus</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Standardising communications</td>
</tr>
<tr>
<td>ARiL9</td>
<td>He has never implemented such a solution though</td>
<td></td>
</tr>
<tr>
<td>ARiL10</td>
<td>Continuous patching of components requires effort</td>
<td>Patching</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Requiring effort</td>
</tr>
<tr>
<td>ARiL11</td>
<td>Therefore, the more components used the more patching effort (and testing) required</td>
<td>Increasing numbers of components</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Patching effort</td>
</tr>
<tr>
<td>ARiL12</td>
<td>However, with certain trusted COTS products, such as Norton Antivirus, there is no need to spend time testing the functionality of the product.</td>
<td>Selecting trusted components</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reducing testing effort</td>
</tr>
<tr>
<td>ARiL13</td>
<td>The only testing is whether the product’s integration works.</td>
<td>Reducing testing effort</td>
</tr>
<tr>
<td>ARiL14</td>
<td>The same concept was used with PcAnywhere – integration testing was performed, not testing of the product.</td>
<td>Reducing testing effort</td>
</tr>
<tr>
<td>ARiL15</td>
<td>The scale of the COTS products used. For example, smaller COTS products, such as Norton AV is much easier (cheaper) to manage than much larger components, such as EMC’s Network Management product.</td>
<td>Selecting smaller components</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reducing effort</td>
</tr>
<tr>
<td>ARiL16</td>
<td>With the latter the vendor struggled with providing support on the integration of this</td>
<td>Poor vendor support</td>
</tr>
<tr>
<td>CodeNo.</td>
<td>Key Text</td>
<td>Early Open Codes</td>
</tr>
<tr>
<td>---------</td>
<td>----------</td>
<td>------------------</td>
</tr>
<tr>
<td>ARiL17</td>
<td>The number of different systems on which the solution has to be integrated affects the maintenance costs.</td>
<td>Maintenance cost driver</td>
</tr>
<tr>
<td>ARiL18</td>
<td>It is important to get the vendor involved early – vendor support is very important in Gaining vendor support.</td>
<td>Aiding integration</td>
</tr>
<tr>
<td>ARiL19</td>
<td>1) getting the product to work in the first place.</td>
<td></td>
</tr>
<tr>
<td>ARiL20</td>
<td>2) keeping the product working over time</td>
<td>Aiding maintenance</td>
</tr>
<tr>
<td>ARiL21</td>
<td>The main variables which affect the TCO of a CBS are: The complexity of the components used.</td>
<td>Cost driver</td>
</tr>
<tr>
<td>ARiL22</td>
<td>The complexity of the system as a whole</td>
<td>Component complexity</td>
</tr>
<tr>
<td>ARiL23</td>
<td>The number of bugs in both the COTS components and any bespoke code.</td>
<td>Code quality</td>
</tr>
<tr>
<td>ARiL24</td>
<td>ARiL24 considers the following to be classed as maintenance activities, and thus contribute to maintenance costs.</td>
<td>Maintenance cost driver</td>
</tr>
<tr>
<td>ARiL25</td>
<td>Patch management – to fix bugs or security vulnerabilities</td>
<td>Managing patches</td>
</tr>
<tr>
<td>ARiL26</td>
<td>System enhancements</td>
<td>Enhancing systems</td>
</tr>
<tr>
<td>ARiL27</td>
<td>Business driver changes</td>
<td>Changing requirements</td>
</tr>
<tr>
<td>ARiL28</td>
<td>Dependencies – For example, following the requirement to upgrade the OS the application may need to be recompiled.</td>
<td>Degree of dependency</td>
</tr>
</tbody>
</table>

Initial memos: Memo (Chris Latham – Integration Architect - Interview)

**Concept Degree of dependency**

Overlapping functionality ARiL1, Investigating dependencies ARiL2 and Underlying dependencies ARiL3

When integrating components underlying dependencies, caused by overlapping functionality, can result. Thus, the requirement for investigating dependencies between components to gain an understanding of the effect of change on system stability is a cost driver because of the amount of effort and skill required to perform these tasks.

With overlapping functionality there may be underlying dependencies which are only understood following tasks to investigate dependencies between components and the effect of change on system stability (for example, making a change to one component sharing functionality with components could have unpredictable results).

Therefore, the effect of change of components on system stability can be understood in terms of the degree of dependency between components. Thus, with a higher degree of dependency between components the greater the effect of change on system stability (and a greater effect on cost resulting from additional effort required resolve problems).

By investigating dependencies of systems containing underlying dependencies system maintainers can gain and understanding of the effect of change on a system.
be seen as a cost reducing strategy because by gaining an understanding of the effect of change upfront allows system developers to plan change activity. Conversely, by not investigating dependencies system developers would be left to react to any unpredictable results occurring from changing system parts.

Concept **beyond sphere of influence**
Re-prioritising tasks ARiL4

*Re-prioritising tasks ARiL4* can be **beyond the sphere of influence** of the people whose tasks are being re-prioritised. Normally, within an organisation a management team allocate tasks to be performed. They invariably have the choice over when tasks are re-prioritised. An effect of *Re-prioritising tasks ARiL4* can be wasting effort if time was originally spent on tasks and following Re-prioritising the tasks are dropped. Even if tasks are resumed at a later date it normally takes additional time understand what actions need to be performed.

*Re-prioritising tasks ARiL4* can be explained with reference to Glaser’s (1978) Six C’s theoretical coding family:

- Tasks allocated → can be affected by condition: requirement to *Re-prioritise task*
- → can have consequence wasting time

A consequence of staff/skill shortages can be the need for *Re-prioritising tasks ARiL4*.

Concept **Staffing cost**
Increasing numbers of components ARiL5, ARiL11, Requiring more staff ARiL5 and Staffing cost ARiL6

There is a link between codes Increasing numbers of components ARiL5 and Requiring more staff ARiL5 in that normally with more components included in a COTS-based additional staff, with different skills, can be required to support a larger variety of components. The skills required to integrate, configure and maintain COTS components can be specialised, and thus cost more. Therefore, the staffing cost associated with implementing many different components could be higher then the staffing costs related to using fewer larger components.

Therefore, with reference to Glaser’s (1978) Six C’s theoretical coding family the following relationship can exist:

- The consequence of *Increasing numbers of components* → can be increasing *Staffing costs* as a result of *Requiring more staff* to support greater numbers of different components.

Concept **Increasing effort**
Patching ARiL10, Increasing numbers of components ARiL5, ARiL11 and Patching effort ARiL11

The link between these codes implies that with CBD there are many tasks *Requiring effort. Patching ARiL10* of components, which normally is a requirement for components throughout their life, is an example. Associated with patching is the effort required to test components following patching. Thus, patching does not normally occur in isolation of testing, because once a component has been patched it needs to be tested to ensure that: a) it works on its own; and b) it works with other components.
With increasing numbers of components ARiL5 the expectation is that patching effort ARiL11 will increase because a consequence of more components is more patching incidents, thus requiring more patching and testing effort. Therefore, increasing numbers of components ARiL5, ARiL11 and patching effort ARiL11 contribute to increasing effort.

Increasing numbers of components ARiL5, ARiL11 → results in increasing patching effort ARiL11 → contributing to increasing effort.

Concept cost reducing strategy and balancing cost challenges
Selecting trusted components ARiL12 and reducing testing effort ARiL12

From the interview data selecting trusted components ARiL12 for use in a CBS can contribute to reducing testing effort ARiL12 because less effort is spent testing the component’s functionality initially and following upgrade/patch activity. However, effort still needs to be used testing the component’s functionality in conjunction with other components and system parts. However, the data did not specify how ‘trusted’ components are identified. The assumption from the data is that the ‘Trusted’ classification results from system developer’s experience of using certain components over different release and patch levels.

Therefore, conceptually the selection of trusted components can be a cost reducing strategy if consciously made with the aim of reducing testing effort.

Unsure if the use of trusted components is a design decision because the term ‘trusted’ is not necessarily associated with architectural standards, and unlike ‘certified’ components there is a degree of subjectivity on the classification of trusted components. Therefore, system developers may be balancing cost challenges of using trusted components to reduce product testing effort with selecting components which do not necessarily support the same architectural standard as other system parts, thus incurring additional integration and testing effort.

Concept establishing relationships
Gaining vendor support ARiL18, aiding integration ARiL19 and aiding maintenance ARiL20

The link between these codes is concept establishing relationships because component customers establishing effective relationships with vendors can contribute to aiding integration ARiL19 and aiding maintenance ARiL20 as the customer and vendor are more likely to work effectively together to resolve integration and maintenance problems quickly. Furthermore, vendors are likely to benefit from additional business resulting from positive reports of good service arising from establishing good relationships.

It can be seen that establishing relationships can be measured in terms of degree (Glaser, 1978). From the data the assumption is of establishing effective relationships, which can contribute to reducing costs by aiding integration ARiL19 and aiding maintenance ARiL20 activities. However, at the other end of the continuum vendors or customers establishing ineffective relationships with each other can be detrimental to reducing cost as integration or maintenance problems would be less likely to be resolved effectively.
When customers are consciously establishing effective relationships with vendors, with the aim of reducing costs, it can be seen to be an example of a cost reducing strategy. It is the conscious act which defines it as a strategy (Glaser, 1978).

**Concept support quality**

Poor vendor support ARiL16

This code demonstrates a property of the concept of support quality which can be measured in terms of a continuum ranging from good support quality through to poor support quality.

From the support quality was defined in terms of vendor’s lack of ability in being able to support their components. The vendor had good intentions to provide good quality support but was lacking ability to achieve this.

However, it can be seen that lack of integrity, care etc. could also result in poor quality, in which case the vendor does not have good intentions of providing good quality support.

**Concept Component complexity**

Component complexity ARiL21

From the data component complexity was defined as a cost driver. The implication was that it can be defined in terms of ‘degree of complexity’. Thus, the continuum for component complexity can range from low to high degrees of complexity.

The data implied a cause and consequence relationship between component complexities and cost in that the consequence of a higher degree of component complexity was higher cost. This can be explained because as component complexity increases the number of configuration parameters and options tends to increase as well as the required skill levels to configure, integrate and maintain components.

Thus, with more configuration parameters there can be more settings to get wrong and suffer effects of change following maintenance activity.

**Concept System complexity**

System complexity ARiL21

From the data the notion of system complexity affects the cost of ownership of COTS-based systems. System complexity was defined as the number of parts, the number of connections and the number of dependencies between system parts. Thus, systems performing complex tasks, comprising of many different components, each with many configuration settings and numerous instances of integration code and high degrees of dependencies could be classed as systems with higher degrees of system complexity than systems containing less of instances of these interconnected variables because with more variables to consider more effort and skill may be required to develop the system in the first place, there are more chances of things going wrong following maintenance task and more effort may be required to resolve any issues.
Appendix C:

Focused Memos of Concepts
Appendix C

Focused Memos of Concepts

The following memos are focused memos explaining the concepts which emerged from the interviews. They are displayed in alphabetical order:

Memo on Concept Balancing cost challenges

<table>
<thead>
<tr>
<th>Introduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>From the data BALANCING COST CHALLENGES conceptualises cost trade off actions performed by COTS-system practitioners when developing and maintaining COTS-based systems. From the literature, a proposed benefit of adopting the COTS-based approach is ‘reducing costs’. The interview data implied that one reason for choosing to build systems from COTS components was related to the perceived cost saving potential of this method. However, with the development and maintenance of COTS-based systems there are challenges with managing cost. The practicalities of the COTS-based approach can result in practitioners having to balance different cost challenges in order to produce systems which meet requirements and then to manage their maintenance over the system lifecycle.</td>
</tr>
</tbody>
</table>

It can be seen that one facet this concept is the cumulative effect of individual design principles or actions on cost. Some principles, when considered individually, can contribute to reducing cost, but when combined can conflict and outweigh any individual cost benefits. Furthermore, system developers and maintainers may be forced to select inappropriate products, processes or principles as a result of other pressures in order to make the system function. Therefore, system developers/maintainers may be balancing the cost reducing choices of selecting certain approaches, products, design principles etc. with the cost increasing effects of other decisions.

For example, building systems with fewer components (Using fewer components ARiB18) is considered to be a contributing factor to reducing total lifecycle costs because it is assumed that less effort is required to configure and maintain a lesser number of interfaces. Thus, Using fewer components ARiB18 is deemed to be an approach to address cost challenges of developing and maintaining systems.

However, Using fewer components ARiB18 to build COTS-based systems can result in some components being functionally larger in size, compared with building the same systems from greater numbers of components, where the logical assumption is that some components must be smaller in size in order to deliver the same system functionality. From the interview data it is seen that Using fewer components ARiB18 can result Functional redundancy ARiB3, where more functionality then is required is provided by larger components. A result of using components with Functional redundancy ARiB3 can be additional effort required to disable redundant functionality (Redundant functionality is normally disabled because leaving it intact requires additional memory and other system processes and resources).

The net result can be seen to be system developers balancing cost challenges of reducing integration and ongoing costs by Using fewer components ARiB18 with increasing costs of disabling Functional redundancy ARiB3.
However, from the data it was not clear if practitioners consciously balanced the cost challenges of building and maintaining COTS-based systems by selecting or applying specific features or design principles or whether balancing cost challenges was a consequence of a combination of design principles. The issue is that during system development time practitioners are unlikely to know the cost requirements throughout of COTS components for the life of the system. Therefore, concept balancing cost challenges more likely relates to the application of design best practices for which the assumption is that the uncertainty of some facets of the COTS marketplace will affect cost. For example, the concept of building systems from components supporting the same architectural standard underlies the assumption that such components will require less effort to maintain during the life of a system than components supporting different architectural standards.

From the data the following code comparisons can be seen to link to concept balancing cost challenges:

Comparing *Increasing system development costs PMiA11* and *Integration effort PMiA8*

Assessing component suitability encompassed tasks requiring human effort and skill, thus, incurring financial cost.

Thus, assessing component suitability can be a contributing factor to *Increasing system development costs PMiA11* depending on the amount of effort and skill required to perform these tasks. The aim of practitioners is to lessen the effect of *Increasing system development costs PMiA11*.

*Integration effort PMiA8* is human effort incurred when integrating COTS-components and other system parts. *Integration effort PMiA8* can be a contributing factor to *Increasing system development costs PMiA11* because human effort incurs financial cost. Again, the aim of practitioners is make design decisions resulting in systems supporting system requirements, but which also minimise *Integration effort PMiA8*.

The link between codes *Increasing system development costs PMiA11* and *Integration effort PMiA8* may require practitioners having to make decisions resulting in balancing cost challenges contributing to *Increasing system development costs PMiA11* with costs related to *Integration effort PMiA8* when developing COTS-based systems.

Comparing *System developers reusing COTS components PMiA1* and *Maintaining multiple interfaces PMiA3*

With reference to the data *System developers reusing COTS components PMiA1* and *Maintaining multiple interfaces PMiA3* can be linked to developers balancing cost challenges as a result of the trade-off between the proposed cost reducing benefits of reusing COTS components and additional costs resulting from maintaining multiple interfaces.

Comparing *Saving development time PMiA2* and *Maintaining multiple interfaces*
P Mi A 3

Saving development time PMiA2 and Maintaining multiple interfaces PMiA3 can be linked to practitioners balancing cost challenges resulting from the benefits gained by using COTS components (Saving development time) with additional ongoing costs relating to Maintaining multiple interfaces (if many components are integrated in the system). However, there is no guarantee that components selected to save development time also result in reducing maintenance costs. Therefore, there can be a balance between system developers selecting components which result in saving development time and selecting components which minimise the requirement of maintaining multiple interfaces. A consequence of getting this balance wrong can result in the human effort, and thus cost, required to maintain multiple interfaces can outweigh the cost benefit of Saving development time PMiA2.

Comparing Saving development time PMiA2 and Increasing system maintenance effort PMiA3

From the data codes Saving development time PMiA2 and Increasing system maintenance effort PMiA3 were linked to concept Balancing cost challenges because, on one hand, choosing to use COTS components can enable system developers to save development time by offsetting the time requirement of writing code themselves. However, due to the volatility of the COTS marketplace using COTS components can result in increasing system maintenance effort (compared with custom code) because over time system administrators have to manage the effects of changing components.

Comparing Saving development time PMiA2 and Conflicting maintenance schedules PMiA10

It can be seen that an inverse relationship exists between Saving development time PMiA2, which should result in lowering development costs, and Conflicting maintenance schedules PMiA10, which can lead to increasing maintenance effort as a result of managing the maintenance (upgrade, patching etc.) of different vendor’s products which do not follow the same time schedule. Thus, successful management of these issues entails system developers and maintainers Balancing cost challenges related to saving development time and increasing maintenance cost resulting from conflicting maintenance schedules.

Comparing Saving development time PMiA2 and Assessing component suitability PMiA11

The link between Saving development time PMiA2 and Assessing component suitability PMiA11 can be seen to be Balancing cost challenges because, on one hand, the use of COTS components is perceived by practitioners to result in Saving development time PMiA2 and thus cost. Conversely, Assessing component suitability PMiA11 can contribute to increasing costs as a result of the human effort and skill required to perform these tasks.

Note: From the data code, Assessing component suitability PMiA11 is related to the processes performed by COTS system developers to assess the suitability of prospective components. This process involves two main tasks: verifying if
Comparing **Maintaining multiple interfaces PMiA3** and **Selecting architecturally compatible components PMiA6**

From the data it can be seen that *Maintaining multiple interfaces PMiA3* is rarely a one off task and involves the requirement for ongoing maintenance tasks to be performed throughout the life of a system, thus contributing to ongoing costs.

Conversely, the expectation is that *Selecting architecturally compatible components PMiA6*, will contribute to reducing integration and maintenance costs because architecturally compatible components should be built to operate together, and thus require less effort to integrate and maintain. However, the dependency of this principle is the existence of suitable architecturally compatible components which support system requirements.

From a cost perspective the ability to select a variety of combinations of components can give practitioners the opportunity to provide higher proportions of system requirements from COTS components. However, a consequence of this approach may be the additional effort and cost required to integrate and maintain greater numbers of components and interfaces because there is no guarantee that all interfaces are supplied from *compatible components*.

Therefore, *Maintaining multiple interfaces PMiA3* is independent of *Selecting architecturally compatible components PMiA6* because multiple interfaces can be maintained in isolation of the selection of architecturally compatible components. However, the expectation is for the costs associated with *Maintaining multiple interfaces PMiA3* of components which are architecturally compatible to be less than the costs of maintaining the interfaces of architecturally incompatible components. Therefore, practitioners are balancing cost challenges associated with maintaining multiple interfaces and producing systems from architecturally compatible components.

Comparing **Maintaining multiple interfaces PMiA3** and **Designers minimising system change PMiA15**

From the data *Maintaining multiple interfaces PMiA3* is associated with effort and thus cost. The more interfaces to be maintained the greater the maintenance cost. ‘System change’ incurs cost as a result of the human effort and skill required to effect change. Therefore, minimising change lessens this cost.

Therefore, the link between *Maintaining multiple interfaces PMiA3* and *Designers minimising system change PMiA15* is practitioners balancing cost challenges of building and then maintaining interfaces of systems built from many components, which can satisfy greater proportions of system requirements with cost challenges of designing systems where the requirements for change have been minimised.

Comparing **Functional redundancy ARiB3** with **Component licensing fees**
ARiB11

With fewer, larger components practitioners may be liable for the whole Component licensing fees ARiB11 even if the components contain Functional redundancy ARiB3 which will be disabled.

Therefore it can be seen that practitioners are balancing cost challenges when paying the whole licence fees of fewer large components and then expending additional effort to disable redundant functionality.

Comparing Reducing ongoing costs ARiC15 with Linking cost with change over time ARiC20

It can be seen that there is an inverse relationship between codes Reducing ongoing costs ARiC15 with Linking cost with change over time ARiC20 because implementing change to COTS-based systems normally contributes to increasing ongoing cost, rather than to reducing ongoing costs. This is because change activity requires effort and skill and, in some cases, specialist skills, all of which contribute to cost.

From the data it was stated that COTS-based systems change over time as a result of COTS components, other technologies and system requirements changing. The result is that the cost benefits of COTS-based development may not last long because they can be outweighed by the cost associated with change. Therefore, it can be seen that practitioners are balancing cost challenges of decisions designed to attain reducing ongoing costs with perceived costs associated with change.

Comparing Licensing costs ARiD34

Code Licensing costs ARiD34 contribute to increasing costs for system developers because financial outlay is required. However, from the data (Interview ARiD), system developers can be seen to be balancing cost challenges if choosing components with lower licensing costs, but requiring additional tailoring costs, compared with using more functionally appropriate components which have higher licensing costs.

Comparing Cost attributing factor SDiE1 with Linking factors SDiE2

From the interview (interview SDiE) data the number of components used in a COTS-based system was identified as a cost attributing factor SDiE1. In addition, Linking factors SDiE2 provide reasons for the effect of the number of components used in a COTS-based system (cost attributing factor) on cost. Examples of Linking factors SDiE2 are problems associated with getting vendor support for more components and the costs of maintaining greater numbers of interfaces associated with greater numbers of components. Therefore, the relationship is that a factor becomes a cost attributing factor SDiE1 as a result of Linking factors SDiE2.

The choice to use a set number of components within a system normally results from design decisions made by system developers. However, when considering the number of components (cost attributing factor SDiE1) to be used in a design system developers may be balancing cost challenges arising from using more components with other costs which could occur if failing to support all system requirements or
implementing the system on time.

Comparing Increasing number of components SDiE1, SDiE4 with Vendor homogeneity SDiE5, Internal organisational complexity SDiE6 and Lacking organisational synergy SDiE7

From the data code Vendor homogeneity SDiE5 encapsulates the notion of different components being supplied by the same vendor. From the data (previous interviews PMiA and ARiB) the choice of system developers to build COTS-based systems from components supplied by the same vendor was considered to be a design decision contributing to reducing costs because of the assumption that the components would be designed to integrate together. Thus, from a design perspective sourcing components from the same vendor is seen to reduce the effort required to tailor and integrate components supplied by different vendors. A further assumption from the interview data was that the maintenance effort would be less for components supplied by the same vendor because vendor would test and verify the integration of upgraded components with all of their products.

Sourcing components from the same vendor was also deemed to reduce the development and maintenance costs of building a system from more components because of the assumption that the components were designed to integrate together.

However, from the data (Interview SDiE) some larger vendors producing many COTS components may comprise of a more complex internal organisational structure, whereby different components are developed and supported by different internal departments or teams. From a product support perspective the different departments of some large vendor organisations can appear to customers as virtually different companies, with little synergy and cooperation between departments. Thus, from the customer’s perspective the cost benefits of sourcing components from the same vendor can be challenged by an increasing support complexity manifested by the requirement of dealing with different support groups when needing product support.

Another link between codes Using more components SDiE1, SDiE4 and Vendor homogeneity SDiE5 can be concept balancing cost challenges because, on the one hand, building systems from more components can add cost for system developers due to the increasing cost of developing and maintaining more interfaces. Conversely, the cost of using a greater number of components can be offset by sourcing them from the same vendor because the assumption is that components supplied by the same vendor will integrate together with minimal effort. However, the cost benefit of sourcing the components could be diminished if the vendor’s internal support structure makes it difficult to gain adequate support for the variety of products.
Introduction
There are many DESIGN PRINCIPLES associated with COTS-based system design, such as selecting components which are highly cohesive, but loosely coupled in order to allow components to be changed in isolation of other system parts, designing systems using components supporting the same architectural standards and exploiting the flexibility of the COTS-based design approach by sourcing components from different vendors.

However, a result of the practicalities of building COTS-based systems in the real world can result in designers Balancing design principles in order to produce functioning systems. Thus, the Design decisions underpinning Balancing design principles tend to result from conscious decisions made by system designers. This differs from Conflicting design principles which can occur as a consequence of design decisions. Thus, with reference to Glaser’s (1978, 1998) strategy theoretical coding family Balancing design principles can be seen to be a strategy for handling the design challenges of building COTS-based systems because the deficiencies of one design principle may be balanced by the benefits of other design principles.

However, Balancing design principles can still result in Conflicting design principles because in attempting to balance design principles designers may be left with design principles which conflict. An example of this is where customers force system designers to select inappropriate components. System designers may attempt to balance the affect by applying other design principles. However, the net effect can be a conflict of design principles.

Balancing design principles emerged from the following code comparisons:

Comparing Architects recommending using fewer components PMiA5 and Selecting architecturally compatible components PMiA6

The association between Architects recommending using fewer components PMiA5 and Selecting architecturally compatible components PMiA6 can be seen to result in designers Balancing design principles. On one hand, architects recommend a COTS-based system design principle of using the least number of larger components with the aim of reducing the number of connections between components. In contrast, an alternative architectural design principle indicates that the number of separate components used in design is not the main factor. Selecting components supporting the same architectural standard was deemed more important; the assumption being that components supporting the same architectural standard are more likely to integrate with less effort than with components supporting different architectural standards. However, designers can be required to make design decisions resulting in Balancing design principles because finding fewer, architecturally compatible components, which also support system requirements may not be available. Thus, designers may be forced to balance between choosing greater numbers of components or components which do not support the same architectural standard.

Comparing Architects recommending using fewer components PMiA5 and COTS supplier issues PMiA9
It can be seen that Architects recommending using fewer components PMiA5 is underpinned by a DESIGN PRINCIPLE which aims to contribute to reducing the number of interconnections between components, thus reducing system complexity.

Further design principles relate to the flexibility of a COTS-based design approach in that to provide the functionality to cover system requirements system designers are not forced to source components from the same vendor. However, COTS supplier issues PMiA9 can ensue as a consequence of using components supplied by different vendors. For example, there is no guarantee that components supplied by different vendors will support the same architectural standard; product support complexity can increase as a result of system developers and maintainers having to deal with greater numbers of support organisations, etc.

In comparison, although tending to support the same architectural standard components supplied by one vendor tend to be limited to one area of speciality, which may not provide coverage of system requirements.

However, a dilemma facing Architects is that the benefit of recommending using fewer components, supplied by different vendors, can be outweighed by additional effort and costs integrating and maintaining components which do not necessarily support the same architectural standard.

Therefore, the link between codes Architects recommending using fewer components PMiA5 and COTS supplier issues PMiA9 can be seen to be Balancing design principles because by attempting to adhere to one ‘good practice’ design principle architects may have to balance the use of conflicting design principles resulting from the availability of suitable components.

Comparing Reducing ongoing costs ARiC15 with Customer pre-conception ARiC18

Some customers have a pre-conception to build systems using the COTS-based approach regardless of whether COTS components are suitable or satisfy the requirements. It can be seen that system designers and developers may end up Balancing design principles by designing systems which may not provide the best architectural solution to contribute to Reducing ongoing costs ARiC15, but satisfy customer’s instructions.

Comparing Underlying dependencies ARiF4 with Designing for change ARiF12

A Design principle of COTS-based system design is Designing for change ARiF12, involving using components which are highly cohesive, but loosely coupled. The aim is to provide systems supporting system requirements, whilst allowing components to be changed in isolation of other system parts.

However, the practicality of building CBS can result in system developers and designers Balancing design principles by including components with Underlying dependencies ARiF4 because of the limited availability of loosely coupled components, which also satisfy system requirements. Therefore, system developers can be forced to compromise on the design in order to produce a functioning system.
Memo on Concept Beyond sphere of influence - ARiB23, ARiB24, ARiC16, ARiC21, ARiD14, ARiD15, ARiD16, ARiD21, ARiK6, ARiL4

Introduction

Concept **Beyond sphere of influence** encapsulates different themes. Firstly the nature of relationships between organisations. Thus, the ability of one organisation to acquire a specific level of assistance and cooperation from another organisation may be beyond their sphere of influence because of different cultural and financial reasons. For example, some organisations have a culture of assisting other parties; whereas others may not provide a similar level of assistance unless specified in a contract.

Secondly, variation to a component’s maintenance schedule or lifespan may be **Beyond the sphere of influence** of system developers and maintainers. For example, vendors can impose their own product upgrade cycle on customers or even decide to withdraw a component independently of the wishes of customers. Furthermore, a component vendor may cease trading, which is also beyond the sphere of influence of their customers.

From the data a contributing factor to the success of COTS-based systems can be related to the degree of cooperation existing between organisations. With COTS-based design system developers are reliant upon components which are developed, supplied and maintained by different organisations. Thus, **Vendor commitment ARiB23** to providing product support (resolving bugs, assisting with integration issues, etc.) can be crucial to system developers and maintainers receiving adequate **Vendor support quality ARiB24**. However, the influence on **Vendor commitment ARiB23** and **Vendor support quality ARiB24** can be **Beyond [the] sphere of influence** of system developers if vendors fail to commit to working closely with them. Thus, because a vendor tends not to be part of the same organisation, system developers cannot normally force a vendor to provide a desired level of service.

Similar scenarios can exist within companies. For example, in large organisations internal support teams can appear as different organisations to those outside of the teams. There may be a lack of synergy and cooperation between internal teams.

**Beyond sphere of influence** is related to category **ORGANISATIONAL ISSUES** because from the data this concept emerged from issues associated with differences between organisations. Thus, it can be classified as an issue with organisations. People within one organisation or team tend to cooperate as a result of the collective values of the members of the organisation or team. However, that collective synergy may not exist between organisations.

From the data **Beyond the sphere of influence** emerged from the following code comparisons:

Comparing **Choosing components supplied by same vendor ARiC16** with **Everything changing over time ARiC21**

A consequence of **Choosing components supplied by same vendor ARiC16** is that system developers may be reliant on components supplied from one vendor. Problems can occur if the vendor subsequently goes out of business. **Everything changing over time ARiC21** can encompass vendors ceasing to trade as well as software and hardware changes. The link between the 2 codes is **Beyond sphere of influence**
because System developers can have the freedom of *Choosing components supplied by same vendor ARiC16*. However, the effect of a vendor ceasing trading is normally beyond their sphere of influence.

Comparing **Lacking product understanding ARiD14** with **Poor product training quality ARiD15**

The background of these codes was that the vendor’s and system developer’s internal support teams were *Lacking product understanding ARiD14*. Furthermore, the company’s internal support team were unable to get adequate training on the vendor’s components (the vendor ran educational classes on the components but they were described as poor quality by the attendees). The result, from the system developer’s perspective, was manifested by numerous problems with the components (software quality and configuration issues) during the system development and maintenance phases.

These issues were **Beyond [the] sphere of influence** of system developers because they had no influence over the internal support team’s product knowledge or the product support quality provided by the component vendor.

It was suggested from the interview data (interview ARiD) that the organisational culture of the company resulted in internal departments behaving like separate companies; the consequence being a lack of cooperation and synergy between different departments.

It can be seen that codes Lacking product understanding ARiD14 with Poor product training quality ARiD15 also contributes to **Increasing costs** because of **Increasing effort** incurred by system developers/maintainers managing product problems, pursuing resolution to product problems from vendor and internal support teams and testing proposed solutions to numerous product problems.

Comparing **Internal product support issues ARiD16** with **Staffing shortage ARiD21**

A consequence of Internal product support issues ARiD16 and Staffing shortage ARiD21 is **beyond sphere of influence** of system developers and maintainers because they have no control over the company’s internal support teams staffing levels.

Comparing **Developing vendor cooperation ARiD26** with **Variable vendor response ARiD30**

The attempt of system developers’ Developing vendor cooperation ARiD26 can be met with Variable vendor response ARiD30 as a consequence the perceived degree of importance of the system developer’s organisation to the vendor. Thus, when the system developer’s organisation is not considered important to the vendor than then the quality of vendor cooperation could be less; all of which can be **Beyond [the] sphere of influence** of system developers.
Memo on Concept Change unpredictability - ARiC2, ARiC4, ARiC8

Introduction
From the interview data ‘change’ is a major cost driver for COTS-based system developers and maintainers because it requires human effort. Depending upon the nature of the change activity the requirement may be for people with different skill types, including consultancy and other specialist skills, which may incur greater cost. Change contributes to increasing cost because it can be unpredictable and thus not be factored within initial cost estimates. Therefore, increasing cost can be a consequence of change unpredictability.

The Implications of change can be unpredictability because a change to one part of a system can necessitate further changes to other system parts due to system dependencies. A knock-on-effect of subsequent change requirements cannot always be predicted.

Requirements changing over time relates to system requirements changing throughout the life of a system. The interview data indicates that system requirements normally always change over the system’s lifespan because the business requirements of the system’s owner change. Changing business requirements, which are not always predictable when companies are exploiting new business opportunities, can necessitate changing system requirements.

From the data the following comparison of codes are linked to concept knock-on-effect:

Comparing Implications of change ARiC2 and Requirements changing over time ARiC4

Implications of change ARiC2 and Requirements changing over time ARiC4 can be seen to be associated to the concept Change unpredictability because the implications of change can be unpredictability and requirements changing over time can introduce further unpredictability. Change unpredictability can contribute to increasing cost for system maintainers. The act of ‘Change’ adds cost because it requires effort to assess, initiate and test the change. If change is planned cost can be minimised because the change complexity can be assessed in advance, the right skills brought in and people trained. With some unpredictable changes there may not be time to train people, thus skilled people may need to be brought in at short notice, contributing further to cost.

Comparing Implications of change ARiC2 and Domino effect ARiC8

Codes Implications of change ARiC2 and Domino effect ARiC8 can be seen to contribute to change unpredictability because of the consequence of system dependencies. With dependencies, the implications of changing one part of a system can cause a domino effect, requiring the changing of other system parts. A Domino effect ARiC8 of change activity can not always be predicted, thus can contribute to increasing cost for system developers/maintainers when these costs were not factored into original cost estimates.
Memo on Concepts Component licensing fees - ARiB10, ARiD34 and Staffing cost - ARiL5, ARiL6

Introduction

Component licensing fees relate to more than just the monetary cost of COTS components. Component licensing fees encompasses the intellectual property value of COTS components to vendors. From vendors’ perspective Component licensing fees are the means by which they aim to recover costs and make a profit. Thus, commercial decisions by vendors to develop components in the first place are related to market trends and their perceived ability to sell the components.

Conversely, for system developers Component licensing fees represent predictable costs incurred in lieu of the costs to develop functionality themselves. For system developers to realise cost savings Component licensing fees should be less than the costs to develop functionality supplied by components themselves.

Thus, it can be seen that Component licensing fees contribute to CONTROLLING COST for two reasons:

When a conscious design decision to use COTS components is made it can be viewed as a Cost reducing strategy if the assumption of incurring Component licensing fees is deemed to outweigh the costs of developing the functionality in-house. Furthermore, the use of COTS components and the associated Component licensing fees can enable system developers to deliver a system to market faster as a result of saving the time of producing the functionality in-house. Reducing the time in which systems are brought to market can contribute to reducing cost.

Component licensing fees can contribute to Increasing cost for system developers as this represents cost expenditure. The contribution of Component licensing fees to Increasing cost may differ as each vendor may charge different amounts for their components. Additionally, Component licensing fees may not be a one-time cost but be required throughout the life of a component. Furthermore, there may be nothing to stop vendors from increasing licensing costs at any time. Thus, system developers may be presented with additional costs leaving them with the dilemma of continuing to pay the licensing costs or bear additional costs in identifying or producing replacement functionality.

Component licensing fees emerged from the following code comparisons:

Comparing code: Component licensing fees ARiB11

Component licensing fees ARiB10, ARiD34, ARiB11 can be seen to be a contributing factor to Increasing costs for system developers because COTS component vendors tend to produce components to sell for a profit – component licensing fees are one way of a vendor charging for their products.

Staffing cost - ARiL5, ARiL6

Introduction

Staffing cost encompasses the cost of employing people. In the commercial world employing people incurs cost (In other domains, such as the charity sector, some people may not charge for their time). Furthermore, the Staffing cost can vary as the costs of employing people with specialist skills tend to be higher than the employment
costs for those with common skills.

Therefore, with reference to Glaser’s (1978; 1998) theoretical coding families it can be seen that Staffing cost is a contributing factor to Increasing costs. This relationship can be defined in terms of degree in that the higher the Staffing cost the greater the contribution to Increasing cost.

Staffing cost emerged from the following code comparisons:

Comparing Increasing numbers of components ARiL5, ARiL11, Requiring more staff ARiL5 and Staffing cost ARiL6

There is a link between codes Increasing numbers of components ARiL5 and Requiring more staff ARiL5 in that normally with more components included in a COTS-based additional staff, with different skills, can be required to support a larger variety of components. The skills required to integrate, configure and maintain COTS components can be specialised, and thus cost more. Therefore, the staffing cost associated with implementing many different components could be higher than the staffing costs related to using fewer larger components.

Therefore, with reference to Glaser’s (1978) Six C’s theoretical coding family the following relationship can exist:

The consequence of Increasing numbers of components \( \rightarrow \) can be increasing Staffing costs \( \rightarrow \) as a result of Requiring more staff to support greater numbers of different components.
Introduction

Conflicting business motives incorporates the conflict of business motives which can occur between different organisations. For example, there can be a conflict between the business motives of COTS component vendors and buyers of COTS components (system developers). COTS component vendors aim to profit by developing and selling components. However, when vendors may withdraw components if they do not have sufficient customers for that product (keeping a component on the market requires investment on their part. They may need to fix problems, produce patches, evolve it etc.). However, customers of component vendors, system developers, purchase components to solve business problems, making their profit from the systems they develop (either by developing a system on commission or selling the system etc.). Therefore, a consequence of vendors withdrawing components, which are critical to the functioning of a system, can be problems for system developers and maintainers.

Conflicting business motives can be applied to other business domains where one organisation aims to purchase a commodity from another. In turn, the buyer’s aim is to use the commodity to further their business aims. The seller endeavours to get the highest price, whereas the buyer’s aspiration may be to acquire the commodity at the lowest price, as well as gaining assurance of continuance of supply.

Conflicting business motives relates to ORGANISATIONAL ISSUES because of differing business motives normally occur between different organisations.

The consequence of Conflicting business motives for one party can be Beyond [the] sphere of influence of other organisations. Thus, if a vendor decides to withdraw a critical component from the marketplace system developers and maintainers may not be able to influence this decision.

From the data the following code comparisons indicated the emergence of concept Conflicting business motives:

Comparing Commercial viability of components ARiC13 and Attributing importance of business critical components ARiC14

A consequence of the association of vendors identifying Commercial viability of components ARiC13 and system developers and maintainers Attributing importance of business critical components ARiC14 can be seen to be concept Conflicting business motives.

There can be a conflict between the business motives of COTS component vendors and buyers of COTS components (system developers). COTS component vendors aim to profit by developing and selling components. However, when vendors may withdraw components if they do not have sufficient customers for that product (keeping a component on the market requires investment on their part. They may need to fix problems, produce patches, evolve it etc.). However, customers of component vendors, system developers, purchase components to solve business problems, making their profit from the systems they develop (either by developing a system on commission or selling the system etc.). Therefore, a consequence of vendors
withdrawing components, which are critical to the functioning of a system, can be problems for system developers and maintainers.

Comparing Poor product quality ARiK6 and Vendor forcing upgrade ARiK6

The effect of Poor product quality ARiK6 of COTS components and Vendor forcing upgrade ARiK6 of components is often be Beyond the sphere of influence of system developers and maintainers (the vendor’s customers) because it is vendors who are responsible for the quality and maintenance of their products, not the vendor’s clients who have purchased the components.

The problem is that there can be Conflicting business motives between vendor and customer. The aim of the vendor is to make a profit from selling identical copies of a component. This can also involve providing fixes to groups of software defects in patches or component upgrades, which tend to be released at specific times. This process may help to keep vendors costs down as they can plan for fixes (or improved features) they need to produce between patches or upgrades. However, the motives of customers are to purchase products, which functions correctly, with which to incorporate into systems, which in turn are (normally) used to generate profit.

With reference to Glaser’s (1978) ‘degree’ theoretical coding family there may be different degrees of influence between vendor and customer to get software defects fixed or to modify the product upgrade cycle. However, this appears to be dependent upon several factors, such as, the status of the vendor within the COTS marketplace, the status of the customer in the eyes of the vendor or the flexibility and determination of the vendor’s organisation to respond to individual customer requests. For example, some vendors, who are considered to be industry leaders within their field of products, may be less willing to fix specific software problems for individual customers. They may make customers wait until for the scheduled software patch to be developed, which may take time to materialise. Conversely, vendors who are keen to extend the supply of components to specific customers may be willing to be more flexible.
Memo on Concept Conflicting design decisions - PMiA2, PMiA18, PMiA19, ARiB2, ARiB4, ARiB14, ARiB13, SDiE1, SDiE4, SDiE8, ARiF4, ARiF12

Introduction

Conflicting design decisions encompasses the notion of Design decisions, which when considered individually can relate to sound design principles, but when implemented together can result in conflicting consequences. For example, the intended consequence of one design decision may be to reduce development costs. The assumed consequence of another decision may be reduced maintenance costs. However, the combination of decisions may result in increasing ongoing cost due to underlying dependencies between components and system parts. Conflicting design decisions can occur as a result of the practicalities and pressures for COTS-based system designers to produce working systems.

A consequence of Conflicting design decisions can result in Conflicting design principles when the result of one decision leads to a conflict of design principles; the benefit of one decision may be outweighed by other decisions.

Designers making Conflicting design decisions may be consciously Balancing design principles in order to produce systems which work and satisfy system and customer requirements.

Conflicting design decisions emerged from the following code comparisons:

Comparing Saving development time PMiA2 and Appreciating system dependencies PMiA18 and System dependencies PMiA19

From the data codes Appreciating system dependencies PMiA18 and System dependencies PMiA19 explain that dependencies can exist between system parts, such as COTS components, operating systems, applications and legacy system parts, etc. Example System dependencies PMiA19 are where component versions only work with specific operating system release levels.

Therefore, an understanding of system dependencies is required by system developers and maintainers because changing one system part, such as applying a patch to the operating system, can cause other system parts to fail or function incorrectly.

Conflicts can exist because the cost benefits accrued from Saving development time PMiA2 can be offset by higher maintenance effort and costs if resultant system dependencies were not considered in the original system design.

Therefore, codes Saving development time PMiA2, Appreciating system dependencies PMiA18 and System dependencies PMiA19 can be linked to concept conflicting design decisions because a consequence of design decisions made with the aim of Saving development time PMiA2 can be increased system maintenance time due to the complexities if System dependencies PMiA19.

Comparing Poor component quality ARiB2 and Customer requesting component ARiB4

The concept of conflicting design decisions can be seen to link codes Poor
component quality ARiB2 and Customer requesting component ARiB4 because system designers would be unlikely to select unreliable components if not specifically requested by customers.

This was an example of a system designer implementing conflicting design decisions because, on the one hand, he would not normally recommend this component because of its poor quality. However, he was obliged to use the component because of customer coercion; the customer had requested the component. Therefore, the designer was reconciling the decision to select an unreliable component with customer requirements.

Comparing Customer requesting component ARiB4 with Incorporating fewer components ARiB13

The choice for Incorporating fewer components ARiB13 in a system can be seen as a design decision aimed at reducing system complexity. However, when customers, rather than architects, choose components Conflicting design decisions may ensue because architects can be forced to implement inappropriate system designs comprising of additional components to accommodate customer selections.

Comparing Increasing number of components SDiE1, SDiE4 with Developing additional interfaces SDiE8 and Maintaining additional interfaces SDiE8

A consequence of an Increasing number of components SDiE1, SDiE4 can be Increasing effort resulting from a requirement of Developing additional interfaces SDiE8 because with greater numbers of components more effort can be required to develop and maintain additional interfaces.

Comparing Increasing number of components SDiE1, SDiE4 with Developing additional interfaces SDiE8 and Maintaining additional interfaces SDiE8 can also be seen to be linked to system developers making Conflicting design decisions when selecting a design involving more components (which may satisfy more of the system requirements than an alternative design involving fewer components), but incurring additional integration and maintenance costs as a result of additional interfaces required to connect greater numbers of components together.

Comparing Underlying dependencies ARiF4 with Designing for change ARiF12

A principle for COTS-based system design is Designing for change ARiF12, involving using components which are highly cohesive, but loosely coupled. This can facilitate systems supporting system requirements, whilst also allowing components to be changed in isolation of the rest of the system. However, a consequence of selecting components with Underlying dependencies ARiF4 can be conflicting design decisions, thus preventing changing components in isolation of other system parts.

The link between the 2 codes can also be conflicting design principles because building COTS-based systems with Underlying dependencies ARiF4 goes against the principle of Designing for change ARiF12.

However, the practicality of building COTS-based systems can result in system developers balancing design principles to include components with Underlying
dependencies ARiF4 because the available selection of loosely coupled components, which also satisfy the system requirements, may not exist. Therefore, system designers can be forced to compromise on the design in order to produce a functioning system.
Memo on Concept Conflicting design principles (as a consequence) - PMiA3, PMiA4, PMiA5, ARiC15, ARiC19, ARiF4, ARiF12

Introduction

There are many COTS-based system design principles. However, the basis of concept **Conflicting design principles** is that the benefit of one design principle may outweigh or result in a conflict when combined with other design principles.

Thus, **Conflicting design decisions** can result in **Conflicting design principles**. However, conscious design decisions can be made whereby **Conflicting design principles** result in **Balancing design principles**. The practicalities of the real-world can require system designers to make conscious decisions resulting in balancing design principles in order to build functioning COTS-based systems with reference to the system requirements and available components.

**Conflicting design principles** emerged from the following code comparisons:

Comparing **Maintaining multiple interfaces PMiA3** and **Preferring large components PMiA4**

**Conflicting design principles** can be seen to explain the link between codes **Maintaining multiple interfaces PMiA3** and **Preferring large components PMiA4** because the assumption of the former code is that increasing maintenance cost could occur as a consequence of the increased complexity of systems constructed from many components. However, building systems from greater numbers of components can be seen to be a design principle which aims to provide more coverage of system requirements using COTS components. For example, it may not be possible to configure larger components to provide support for the requested system requirements. Conversely, combining a selection of smaller specialist components may support greater proportions of system requirements. The latter code, **Preferring large components PMiA4**, implies a design principle aimed at reducing system complexity by building systems from fewer, larger components.

Comparing **Maintaining multiple interfaces PMiA3** and **Architects recommending using fewer components PMiA5**

There can be a conflict between **Maintaining multiple interfaces PMiA3** and **Architects recommending using fewer components PMiA5**; the underlying assumption motivating system designers to recommending using fewer components in a COTS-based system design, thus reducing system complexity. This conflicts with maintaining multiple interfaces, where ‘multiple interfaces’ equates to the integration of many components.

Thus, **Architects recommending using fewer components PMiA5** can be viewed as a design principle which reduces system complexity. Conversely, as a design principle, building systems from more components can result in greater proportions of system requirements being supported from COTS functionality. However, a consequence of using more components can be increasing system complexity and a requirement for **Maintaining multiple interfaces PMiA3**, resulting in additional maintenance effort. Therefore, both codes can be linked to the concept of **conflicting design principles** – fewer components reducing system complexity but not necessarily supporting all
Comparing **Reducing ongoing costs** ARiC15 with **Customer expectation** ARiC19

When customers insist on using specific COTS solutions, when not deemed suitable, recognised design principles (such as building COTS-based systems for maintainability, change etc.) may not be able to be adhered to. As such, customers may not achieve the perceived cost savings of adopting the COTS-based approach. Therefore, system designers may be forced to implement and manage **Conflicting design principles** because when producing designs to satisfy **Customer expectations** ARiC19 of using a COTS-based approach, a consequence of **Reducing ongoing costs** ARiC15 may not be achievable.

There is therefore a requirement for managing customer expectations.

Comparing **Underlying dependencies** ARiF4 with **Designing for change** ARiF12

A COTS-based system design principle is **Designing for change** ARiF12, involving using components which are highly cohesive, but loosely coupled. The aim is to facilitate systems supporting requirements, whilst also allowing components to be changed in isolation of the rest of the system. However, **Underlying dependencies** ARiF4 between components can conflict with this principle because it can prevent changing one component in isolation of other system parts.

The link between the 2 codes can be seen to be an example of concept **Conflicting design decision** because design decisions resulting in underlying dependencies conflict with design decisions aiming to apply the design principle of designing for change.

A consequence of **Underlying dependencies** ARiF4 and **Designing for change** ARiF12 can also lead to **Conflicting design principles** because building COTS-based systems with underlying dependencies conflicts with the design principle of designing for change.

However, the practicality of building CBS can result in system developers **Balancing design principles** to include components with underlying dependencies because the available selection of loosely coupled components, which also satisfy system requirements, may not exist, therefore, system developers can be forced to compromise on the design in order to produce a functioning system.

Comparing code: **Requiring external certification** ARiB4

A consequence of acquiring components **Requiring external certification** ARiB4 can contribute to **Conflicting design principles** when they are forced upon system designers and do not fit in with the architectural basis of the rest of the system.
Memo on Concept Cost reducing strategy

Introduction
From the data concept *Cost reducing strategy* emerged as an underlying process employed (or considered) by software practitioners (IT architects, Project managers etc.) of COTS-based system design and management. The commercial challenges and pressures of the environment in which the practitioners operate force them to employ cost reducing strategies. Reducing costs appears to be a common aim for all of the interviewees, whether they are IT architects, project managers or software support personnel.

Furthermore, from the data the notion of cost was referred to by two categories: The cost to develop systems; and the ongoing cost to maintain systems. Although, different actions were taking place during development or maintenance phases of projects practitioners the commonality was that they still adopted *cost reducing strategies*. However, some cost reducing strategies affecting maintenance cost had to be thought about during the development phase of a system because they involved system design decisions which would be difficult to implement once a system was implemented.

Concept *Cost reducing strategy* comprises of three parts: the concepts of *cost*, *reducing* and *strategy*.

On *cost*, although the final measure may be a financial, monetary value the concept is used interchangeably with other concepts, such as, *effort, time* or *requiring skill*. The assumption from the data is that these concepts influence the final monetary value.

*Effort*, for example, relates to human effort, which in turn equates to a financial value: employee or contactors salary. Thus, more people working on a task contribute to extra cost.

*Time* has similar properties to effort in that the longer people spend working on a task the higher the financial cost due to the additional monetary value spent on salaries. However, *saving time* can have additional benefits over just financial saving in wages because in many instances delivering a project to an agreed timescale avoids late delivery financial penalty charges.

*Requiring skill* relates to the skills that people performing tasks must possess. Some tasks are more specialised than others, thus requiring people to hold specialist skills. However, the cost of employing people with specialist skills is normally higher than the costs to employ people with general skills. Therefore, employing a person with specialist skills over a longer period of time has a greater effect on cost than employing a person with general skills over the same period of time.

From the data the concept of *reducing* indicates one aim of software practitioners which, in this case, is *reducing cost*. However, the aim of reducing *cost* normally requires some sort of action which practitioners hope will result in reducing cost – i.e. reducing cost does not just occur in isolation of the decision to perform an action. Thus, to instigate *reducing cost* software practitioners *consciously* select strategies (tactics, techniques), which they believe will result in *reducing cost*. Glaser (1978) stated that the defining quality of a *strategy* is whether it is *consciously* selected. He explained that if an action is not consciously selected it is merely a *consequence* of
Glaser’s (1978) ‘Six C’s’ theoretical coding family, which comprises of the following codes: causes, contexts, contingencies, consequences, covariance’s and conditions, can explain the relationship between the constituent parts of concept Cost reducing strategy (cost, reducing and strategy) for the following reasons:

As indicated above, an action is defined as a strategy if consciously selected with the expectation of achieving a specific end goal. In the case of a cost reducing strategy the intended end goal is reducing cost. Thus, with reference to Glaser’s ‘Six C’s’ coding family (1978) strategy is the cause and reducing cost the intended consequence. The success of the cost reducing strategy assumes that suitable conditions exist (or the absence of conditions resulting in rising costs).

However, as can be seen in the analysis of the data in this study not all conditions can be predicted or controlled. There are cases where software practitioners consciously select a cost reducing strategy believing that all conditions have been accounted for when something untoward occurs, which was not be predicted. The consequence of an unpredicted event can be increasing costs. For example, one design principle is for IT architects to build COTS-based from as few, larger components as possible. This is the strategy because it is a conscious decision with the belief that integration costs will be reduced due to a saving in effort in integrating fewer components. One assumption is that other conditions, such as the necessity to spend additional effort disabling redundant functionality of larger components, is not required.

**Need for a Cost Reducing Strategy**

From the data it appears that the requirement for cost reducing strategies arises as a consequence of commercial pressures and competition experienced by IT companies, such as IBM, to reduce costs.

From a cost perspective there are different phases of activity which have different cost implications. Firstly, during system development phases the cost challenges are to develop systems which adhere to customer requirements, are implemented within agreed time frames and kept within estimated costs.

Secondly, once implemented there are cost challenges to keep systems running. A contributing factor to ongoing costs of COTS-based systems is change. Ongoing costs can arise as a result of managing change instigated by component vendors, operating system upgrades and patching, hardware changes and changes in system requirements initiated by customers.

If the system is handed over to the customer following implementation then the customer normally assumes these costs. However, when IBM manages systems on a customer’s behalf then IBM will normally be responsible for ongoing costs within the constraints of the contract between the organisations.

The properties of concept Cost Reducing Strategy in this study is explained with reference to the data and Glaser’s (1978) Theoretical Codes.

**System Development Costs**

Building systems from COTS components can save development time because
Software developers avoid the time spent producing the functionality supplied by the COTS software components; this effort has already been performed by the developers of the COTS components. Therefore, the thought decisions underpinning code can be seen to be a Cost Reducing Strategy because system developers assume that the result will be Saving development time. Cost Reducing Strategy is a strategy because the choice to build systems from COTS components, as opposed to selecting other development methods such as the custom-built approach, is consciously made. Glaser (1978) defines a strategy as a tactic or a means of dealing with events. Thus, the choice of using COTS software components can be seen as a conscious tactic employed by system developers with the aim of saving development time. Saving development time is assumed to contribute to reducing costs because of a saving in the cost of employing human resources.

Furthermore, the Cost Reducing Strategy of building systems from COTS components can be seen to support a ‘causal-consequence’ model (Glaser, 1978). With the comparison of codes and Saving development time PMiA2 a causal factor is the economic pressures experienced by IT companies of continually reducing system development costs. Thus, the consequence is System developers using COTS components PMiA1, the choice of a system development method promising development cost savings.

From the interview data (xxxx interview) the commercial pressures for IT companies to remain competitive has resulted in the COTS-based system development method being perceived by system developers as the cost effective method of choice for building systems.

Preferring large components PMiA4 in COTS-based system design can be seen as a cost reducing strategy, because it is a conscious design consideration, employed by system developers with the aim of Saving development time PMiA2.

From the interview data, it was stated that varying amounts of effort and time can be required when integrating components. The modification of interfaces required to enable components to communicate together can be a significant source of effort. Therefore, System developers Preferring large components PMiA4, compared with choosing more, smaller components, contributes to Saving development time PMiA2 because less effort is required to integrate a lower number of interfaces.

Selecting architecturally compatible components PMiA6 can be considered a cost reducing strategy because it is a conscious tactic employed by system developers aimed at Saving development time PMiA2.

From the data codes Selecting architecturally compatible components PMiA6 and Avoiding architectural incompatibility PMiA7 relates to the integration of COTS components without the need for interface modification. For example, a problem with using components supporting different architectural standards can be the incompatibility of data formats supported by each interface. To integrate these components requires modification of the data flow between components which due to the specialist nature and unpredictability of the tasks can require considerable effort and specialist skills to modify the data flow between components.
Conversely, with components built around the same architectural standard less integration effort tends to be required because the interfaces support compatible data formats.

Standards are a way of reaching agreement among interacting participants. A standard establishes uniform engineering or technical specifications, criteria, methods, processes or practices (Simanta, Lewis, Morris & Wrage, 2008).

Furthermore, an underlying assumption of codes Selecting architecturally compatible components PMiA6 and Avoiding architectural incompatibility PMiA7 was that the number of interconnected components was not the main cost driver. Integrating a greater number of components of the same architectural standard can require less effort than integrating fewer components built from different architectural standards because in the former case effort to modify the interfaces may not be required.

The basis of code Multiple vendors PMiA10 relates to the concept of a vendor, which is a company, organisation or person selling something, normally for profit, which in the case of this study vendors are selling COTS components. However, one vendor may not be able to supply a selection of components which would satisfy all of the requirements of a system developer.

Code Multiple vendors PMiA10 explains that COTS-based systems can be built from components supplied by more than one vendor. The ability to develop systems this way forms the basis of the COTS-based design approach. COTS-based design has similarities to traditional engineering techniques, such as civil engineering, in which the construction of a bridge, for example, requires a set of components (nuts, bolts, metal sections etc.) but does not necessarily require them to be supplied by the same supplier.

Therefore, acquiring components from Multiple vendors PMiA10 can be a conscious choice made by system developers because this offers an opportunity of supplying greater proportions of system functionality from COTS-components. The proposed consequence is Saving development time PMiA2 because of the time and effort saved from not producing any missing functionality from scratch. However, to achieve saving development time desired conditions should exist, such as all components supporting similar architectural standards, reducing the need for some time consuming tasks required when integrating disparate components.

However, from this analysis there appears to be a contradiction between codes Sourcing components from same vendor ARiD1 and Multiple vendors PMiA10 because both codes can be linked to concept cost reducing strategy when they relate to conscious decisions made by software practitioners. Code Sourcing components from same vendor ARiD1 encapsulates acquiring components from the same vendor because the assumption is that components supplied by one vendor will support the same architectural standard. A further assumption is that each component supplied by the same vendor will be built to integrate with each other with minimal effort. However, there is no guarantee that one vendor’s component selection will satisfy all system requirements. Conversely, as discussed above, code Multiple vendors PMiA10 implies the ability to source a variety of components from more than one vendor to address shortfalls in system requirements compared with the limited set of components supplied by one vendor. However, as a cost reducing strategy the link
between codes Sourcing components from same vendor ARiD1 and Multiple vendors PMiA10 is for the components to support the same architectural standard, where the integration effort is likely to be less than integrating components built around different architectural standards.

Codes Sourcing components from same vendor ARiD1 and Acquiring pre-integrated components ARiD9 can be linked to the concept Cost reducing strategy because system developers may choose pre-integrated components to avoid human intervention and the associated costs of integrating sets of individual components. The expectation of Acquiring sets of pre-integrated components from the same vendor assurance that each component set will have been built and tested to function, thus resulting in lower costs compared with the perceived costs of integrating individual components supplied by different vendors.

Selecting components built around a common architecture AFiF1 is a Cost Reducing Strategy employed by system developers for aiding component integration ARiF2 because the assumption is that each component will include a compatible interface requiring little modification during integration. Thus, the desired consequence of acquiring components supporting a common architecture is reduced integration costs.

The choice of Incorporating fewer components ARiB13 into COTS-based systems can be seen to be a cost reducing strategy employed by system developers, aimed at reducing component installation effort. The assumption is that fewer components require less integration effort, such as, glue code production, writing of integration scripts etc.

Ongoing Costs
With reference to Glaser’s (1978) strategy coding family the pattern linking codes can relate to concept cost reducing strategy when arising from conscious decisions made by software practitioners who believe that consequence will be reducing ongoing costs.

Compared with Reducing ongoing costs ARiC15 code Selecting fewer components ARiC15 can be a cost reducing strategy because the assumption is that the consequence of selecting fewer components is reduced system complexity due less effort being required to manage a lower number of interfaces throughout the life of a system. This contributes to reducing ongoing costs. However, to realise the effect on ongoing costs Selecting fewer components ARiC15 would normally have to be considered during the design phase because replacing a number of components with fewer components after system implementation would probably result in increasing cost depending upon the amount of effort involved.

The link between code Reducing ongoing costs ARiC15 and Choosing components supplied by same vendor ARiC17 can be linked to concept cost reducing strategy because the assumption that components supplied by same vendor support the same architectural standard and are built to integrate together, thus requiring less effort to integrate and to maintain. Therefore, with this in mind system developers, when selecting components supplied by same vendor are employing a strategy for reducing costs: integration costs and ongoing maintenance costs.

Code Choosing pre-integrated components ARiC17 can contribute to Reducing
ongoing costs ARiC15 by Reducing user intervention, thus reducing human effort. Therefore, Choosing pre-integrated components ARiC17 can also be seen as a cost reducing strategy employed by system developers on the assumption that the costs associated with human effort are reduced to integrate and maintain sets of components.

A Statement of compatibility ARiD24 can be issued by vendors confirming that different components will work together. From a cost perspective, by specifically selecting components covered by a Statement of compatibility ARiD24 can be seen to be a cost reducing strategy employed by system developers because components which have been assessed to work together are likely to cost less to integrate, maintain and test, compared with components which do not have this guarantee.

Codes Sourcing components from same vendor ARiD1 and Acquiring pre-integrated components ARiD9 can be a Cost reducing strategy relating to ongoing costs because the assumption by system developers is that vendors will supply upgrades and perform testing confirming that sets of pre-integrated components continue to function. This is specifically true if accompanied by A Statement of compatibility ARiD24 in which the vendor confirms that the components will still function. The ongoing costs to effect changes and upgrade tasks on a combination of individual components are considered to be higher.

Incorporating fewer components ARiB13 (which can be seen to be linked to Preferring large components PMiA4 because the perception is that a large component will provide more system functionality than a smaller component. Therefore, selecting larger components implies that fewer components will be required) into COTS-based systems is a cost reducing strategy which can contribute to reducing ongoing costs because the assumption is that with lower numbers of components there are fewer interfaces to maintain during the life of the system. When compared with Multiple vendors PMiA10 the consequence of Incorporating fewer components ARiB13 on cost is the perception that less effort is required as a result or managing fewer vendor relationships. However, merely equating cost with the number of client/vendor relationships may not fully define the underlying processes. For example, an awkward or uncooperative client/vendor relationship where Denying responsibility for the cause of problems may result in more effort spent resolving issues than in the case of numerous cooperative client/vendor relationships where the emphasis is in resolving issues in a timely manner.

From a maintenance perspective Incorporating fewer components ARiB13 can be a cost reducing strategy because it is perceived to result in reducing system maintenance costs PMiH6 by Reducing potential problems PMiH6 caused by connecting fewer interfaces.
Memo on Concepts Cultural misunderstandings - PMiA9, PMiA10, PMiA13
Appreciating cultural factors - PMiA2, PMiA13
Lacking common understanding - PMiA2, PMiA16, PMiA17, PMiJ9, PMiJ10

**Introduction**

**Cultural misunderstandings** conceptualises that the consequence of cultural differences can result in misunderstandings. One example of this is the difference in the use and interpretation of language between cultures.

**Cultural misunderstandings** emerged from the following code comparisons:

Comparing **COTS supplier issues PMiA9, Multiple vendors PMiA10** and **Difficulty understanding terminology PMiA13**

The link between codes **COTS supplier issues PMiA9, Multiple vendors PMiA10** and **Difficulty understanding terminology PMiA13** can be **Cultural misunderstandings** because developer’s **Difficulty understanding terminology PMiA13** can be caused by cultural differences of multiple vendors, who, potentially, originate anywhere in the world.

With reference to Glaser’s theoretical coding families the affect of **Multiple vendors PMiA10** on **Cultural misunderstandings** can be measured in terms of **degree**. The consequence of using greater the numbers of components, sourced from multiple vendors, based in different countries, can result in a greater chance of system designers, developers and maintainers misinterpreting the published component functionality as a result of **Cultural misunderstandings**.

The impact of **Cultural misunderstandings** can be alleviated by **Appreciating cultural factors** because by being conscious that cultural factors exist and can affect a parties understanding of others can help address misunderstandings.

**Appreciating cultural factors - PMiA2, PMiA13**

**Introduction**

**Cultural factors** conceptualises that cultural difference can exist between people originating from different countries and their use of and interpretation of language. **Appreciating cultural factors** can assist people in dealing with the affect of cultural differences. For example, with a conscious appreciation cultural factors system designers, developers and maintainers are less likely to accept technical descriptions of components at face value.

**Appreciating cultural factors** emerged from the following code comparisons:

Comparing **Saving development time PMiA2** and **Difficulty understanding terminology PMiA13**

From the data code, **Difficulty understanding terminology PMiA13**, relates to the problems system developers have in assessing the suitability of components when vendors use inconsistent terminology to describe the functionality of their components.
From the data vendors publish details of the functionality of their components. However, due to a variety of reasons, such as their country of origin or other cultural factors, there is an inconsistently in the way different vendors describe the capabilities of their components. This adds difficulty to system developers’ ability to assess component suitability because one vendor may describe component functionality differently from another vendor. Thus, the effect on cost is that system developers have to spend more time establishing how components work. Time relates to cost.

Culture is taken to mean: “the collective programming of the mind which distinguishes the members of one group or category of people from those of another” and can apply to nations, organisations, occupations, and professions (Hofstede, 1994, p4). From the interview data the assumption was that cultural relativity, the culture of the human environment in which an organisation operates (Hofstede, 1994) can influence the level of support customers (in this case, system developers are the customers of component vendors) receive.

Therefore, Saving development time PMiA2 can be achieved by system developers if they address Difficulty understanding terminology PMiA13, when assessing components, by Appreciating cultural factors of different vendors.

**Lacking common understanding - PMiA2, PMiA16, PMiA17, PMiJ9, PMiJ10**

**Introduction**

Lacking common understanding conceptualises that some factors relating to building COTS-based systems may not be understood in the same way by all parties. Lacking common understanding can occur as a result of Cultural misunderstandings because of the way people from different countries and cultures interpret factors.

Lacking common understanding emerged from the following code comparisons

Comparing Saving development time PMiA2 and Problems classifying system maintenance PMiA16 and Lacking common understanding of maintenance PMiA17

Codes, Problems classifying system maintenance PMiA16 and Lacking common understanding of maintenance PMiA17, relate to the differing perception held by system developers on what tasks constitutes ‘system maintenance’, compared with system development or other system lifecycle stages. For example, some developers classify tasks occurring after system implementation as system maintenance whereas others consider changes system parts, whilst the system is being developed, as system maintenance. The result of this can be a massage of monetary figures.

The link between codes Saving development time PMiA2, Problems classifying system maintenance PMiA16 and Lacking common understanding of maintenance PMiA17 is concept Lacking common understanding if software practitioners claim Saving development time by mistakenly classifying system development tasks as maintenance tasks.
Comparing Design principle PMiJ9 and Failing to follow standard PMiJ10

Sourcing components supporting the same standard is considered to be a COTS-based system design principle because the assumption is that these components will integrate (with other components and system parts) with minimal effort and result in less effort when maintained.

However, from the interview data details on what constitutes the same standard varied. In some instances the same standard was taken to mean ‘supporting the same API’. In other instances it was taken to mean supporting the same architectural standard. However, for the purpose of this study supporting a common standard is taken to mean that there is a similarity in the components which facilitates their integration and maintenance with minimal effort. Conversely, the integration of components supporting different standards normally requires additional effort, and this cost, in order to manipulate and standardise the communication between components and other system parts.

Codes Design principle PMiJ9, when vendors purport components to adhere to a given standard, can link to the concept Lacking common understanding when the components are found not to support the published standard (code Failing to follow standard PMiJ10). Lacking common understanding relates to the details publicised by vendors and the expectations of their customers when these expectations are not met.

However, from the data no reasons were given for this. It may be surmised that a vendor’s interpretation of the standard may not be the same as that of their customers. There may be cultural differences with the interpretation of the language used by vendors to describe their product’s functionality by customers in other parts of the world. This is an area for further investigation as the consequence selecting components which prove not to function as described can be costly in time and effort. Time and effort can be wasted testing an inappropriate component. Additional time and effort could be spent in locating and testing alternative components or in modifying (if possible) and testing the original component.

Therefore, the following relationship can exist:

Lacking common understanding can contribute to Increasing cost
Memo on Concept Degree of dependency - ARiC1, ARiC2, ARiC3, ARiF4, ARiF5, ARiK8, ARiK7, ARiL1, ARiL2, ARiL3

Introduction
Concept Degree of dependency can be defined in terms of the number of interrelated dependencies between COTS components and system parts and the effect on the system when making changes to one or more system parts.

With reference to Glaser’s (1978; 1998) Degree theoretical coding family Degree of dependency can be measured in terms of a continuum ranging from low through to high Degree of dependency. The higher the Degree of dependency between greater numbers of system parts the greater the contribution to Maintenance complexity and System complexity as a result of more dependent factors to be considered. This is because as Degree of dependency increases the ability to change one system part, in isolation of changing other system parts, diminishes.

Degree of dependency is a factor of MANAGING COMPLEXITY because dependency can contribute to complexity. The higher the Degree of dependency between components, the more factors to be taken into account by those managing complexity.

Degree of dependency emerged from the following code comparisons:

Comparing System dependencies affecting maintenance costs ARiC1 and Implications of change ARiC2

System dependencies affecting maintenance costs ARiC1 explains that there can be an association between dependent COTS-based system parts, contributing to concept degree of dependency, which in turn can contribute to maintenance complexity and maintenance costs.

Implications of change ARiC2 explains that change can have different implications.

The assumption by system maintainers is that greater degrees of system dependencies can result in higher maintenance costs because performing maintenance on one system part can require other system parts to also be maintained. For example, the Implications of change ARiC2, when attempting to upgrade component A, can require component B to also be upgraded as a result of the degree of dependency. Furthermore, Component B may need to be installed on a specific operating system level with middleware products at specific release levels. Thus, depending upon the Degree of dependency, a seemingly small change can result in much more change activity.

Comparing System dependencies affecting maintenance costs ARiC1 and Knock-on effect ARiC3

The concept Degree of dependency can link System dependencies affecting maintenance costs ARiC1 and Knock-on effect ARiC3 because when changing one component the Degree of dependency between the component and other system parts can necessitate a ‘knock-on effect’ of system maintainers needing to upgrade other components in order to preserve system stability.
Comparing Underlying dependencies ARiF4 with Managing component dependencies ARiF5

The Degree of dependency linking the number of Underlying dependencies ARiF4 of different vendors components and the effort of Managing component dependencies ARiF5 when maintaining systems can contribute to ongoing system maintenance costs for system administrators because the greater the degree of complexity (i.e. more underlying dependencies between components can lead to greater time, effort and skill, to assess the effects of component dependencies and the effect of component change on system stability, as well as additional system downtime resulting from additional change activity). For example, the requirement to upgrade one component may necessitate upgrading other COTS components and other system parts.

Therefore, concept Degree of dependency can be defined in terms of the number of interrelated dependencies between COTS components and system parts and the effect on the system when making changes to one or more system parts.

Comparing code: Managing dependencies ARiK7

The effort required in Managing dependencies ARiK7 between components is dependent upon the Degree of dependency between components [and system parts] and can contribute to Increasing cost in two ways:

During system development: The dependencies between COTS components and other system parts have to be understood. This leads to the requirement to install compatible product levels installed in order to initially implement COTS-based systems. A consequence of a higher Degree of dependency between system parts can be additional effort to plan and test that systems function correctly as a whole.

A result of change: There is a link between Degree of dependency, MANAGING CHANGE and managing dependencies because with higher Degree[s] of dependency between system parts, increasing effort can be required in managing dependencies when managing change because change is most likely to disrupt existing dependencies.

The following shows the relationship between Degree of dependency and other factors:

Degree of dependency → affects amount of effort spent required for → managing dependencies → as a result of managing change

Regarding dependencies, it can be suggested that once all dependencies between components have been addressed and the system is functioning as designed managing dependencies would no longer need to occur if nothing changed with the system. It is change which is likely to cause the greatest disruption to dependencies (An example was given in interview ARiC where the degree of dependencies of systems ‘sat in the corner running for years’ supporting business processes which will not change, would have limited effect. This is assuming that the hardware keeps functioning and any licensing charges are paid.)
However, from the interview data ‘constant change’ appears the norm for COTS-based systems, as a result of vendors providing additional functionality, vendors supplying component patches and changing customer requirements.

Comparing code: Providing suitable infrastructure ARiK8

Providing suitable infrastructure ARiK8 to enable COTS components to function correctly can be viewed as a dependency because the dependency for some components is a requirement for specific hardware and network infrastructure solutions. Therefore, the dependencies related to Providing suitable infrastructure ARiK8 can be defined by Degree of dependency. The more dependent COTS components are on specific infrastructure requirements the more factors needing to be considered when Managing change.

Comparing Overlapping functionality ARiL1, Investigating dependencies ARiL2 and Underlying dependencies ARiL3

A consequence of integrating different components can be Overlapping functionality ARiL1 resulting in Underlying dependencies ARiL3. Thus, the requirement for Investigating dependencies ARiL2 between components by system maintainers to gain an understanding of the effect of change on system stability contribute to increasing cost because of the effort and skill required to perform these tasks.

The effect of change of components on system stability can be understood in terms of the degree of dependency between Overlapping functionality ARiL1 of components. Thus, with a higher degree of dependency between components the greater the effect of change on system stability (and a greater effect on cost resulting from additional effort required resolve problems).
Memo on Concept Denying responsibility - ARiD16, ARiD17, ARiD18, ARiD19, SDiE1, SDiE4, SDiE10 and Attributing accountability SDiE1, SDiE4, SDiE9

Introduction

Denying responsibility encapsulates the unwillingness of some organisations to take responsibility for issues relating to their products. This concept can be defined in terms of degree because some organisations may exploit any opportunity to deny responsibility, whereas other organisations may adopt this stance when there is sufficient evidence to support this.

It can be seen that there are situations where Attributing accountability can be inversely related to Denying responsibility. In situations where Attributing accountability is straightforward there is less chance of organisations Denying responsibility. However, where Attributing accountability is challenging Denying responsibility by some organisations may be more likely to occur.

In complex environments problem root cause may not be obvious. This can be further compounded if vendors are denying responsibility for of problems when they feel the problem cause is beyond the scope of their code [normally black box code].

Attributing accountability of problems in complex systems, comprising of more components, greater numbers of interfaces, glue code, wrapper code, etc. may be more challenging because vendors may be unable to locate and isolate the problem root cause of a problem. Furthermore, some vendors are unwilling to cooperate with other vendors (or other departments within the same vendor’s organisation) to aid problem isolation. It could be suggested that attributing accountability of problems in systems comprising of fewer components and other integrated parts can be simpler as a consequence of fewer factors to consider.

However, from the interview data it is not just the number of components which affects the ability of attributing accountability to problems. The number of involved parties and complexities of customer/vendors relationships can also contribute. For example, in systems containing greater numbers of components supplied by the same vendor, where good working relationships exists between customer and vendor and where vendors willingly cooperate to aid problem resolution the challenge of attributing accountability by the customer may not exist because the vendor assumes a degree of responsibility.

Denying responsibility is an ORGANISATIONAL ISSUE as it relates to organisations denying responsibility to problems.

System developers and maintainers may have no influence over other parties denying responsibility. Therefore, Denying responsibility by one organisation can be Beyond [the] sphere of influence of other organisations.

From the data the comparison of the following codes indicate the emergence of Denying responsibility:

Comparing Internal product support issues ARiD16, Differing product support arrangements ARiD17, ARiD18, Failing to take responsibility ARiD17, ARiD18
with ‘buck passing’ ARiD19

The combination of Internal product support issues ARiD16, Differing product support arrangements ARiD17, ARiD18, Failing to take responsibility ARiD17, ARiD18 and ‘buck passing’ ARiD19 can contribute to an organisation Denying responsibility. By buck passing the organisation’s internal support groups were Denying responsibility for the cause and resolution to problems with their components. The consequence is Internal product support issues ARiD16 for system developers/maintainers.

From the data denying responsibility appears to relate to the company’s internal culture because although the different support teams are part of the same company they behave as if separate company boundaries exist.

Comparing Increasing number of components SDiE1, SDiE4 with Buck passing SDiE10

From the data (interview SDiE) Buck passing SDiE10 by vendors is common where they deny that the root causes of problems is with their products. This effect is increased with an Increasing number of components SDiE1, SDiE4 where problem root cause is not easily attributable to specific products and where greater numbers of vendors are involved within problem diagnosis. The consequence is that those vendors may be more willing to pass the buck if they can get a way with it. For example, in a situation where the system is built from only one component the vendor would have less justification for denying responsibility for any problems because their component in the only one making up the system.

From the data the comparison of the following codes indicate the emergence of Attributing accountability:

Comparing Increasing number of components SDiE1, SDiE4 with Isolating problems SDiE9

From a support perspective, the number of components in a system can affect the ability of system developers, maintainers and vendors to isolate problems.

When there is an interaction of different vendors’ products, in-house built glue code, wrapper code and possibly legacy application code Isolating problems can be challenging for customers (system developers/maintainers) and vendors. This can be compounded with systems containing more components because there may be more coding factors, such as interfaces, to consider as well as more customer/vendor relationships to manage. All of which can contribute to increasing support complexity when the customer has to deal with greater numbers of vendors’ support groups or different support teams within the same vendor organisation.
Memo on Concept Design decision - PMiA2, PMiA3, PMiA18, PMiA19, PMiA14, PMiA15, ARiB10, ARiB12, ARiB13, ARiC15, ARiC16, PMiG12, ARiK3, ARiK4, ARiK9

Introduction

A **Design decision** is made by practitioners when designing COTS-based systems. The aim is for Design decisions to support **DESIGN PRINCIPLES**, which are recommendations of how systems should be designed; However, exceptions to this aim can occur when recognised design principles are not possible or suitable.

The concept of a Design decision is not specific to COTS-based design. It can be seen that design decisions, made with reference to design principles, can be applied to any field of design.

**Design decision** emerged from the following code comparisons:

Comparing **Maintaining multiple interfaces PMiA3** and **Designing for change PMiA14**

From the data it can be seen that a consequence of **Designing for change PMiA14** can reducing the effort, and thus cost, of **Maintaining multiple interfaces PMiA3** when the design reflects the design principle and requirement for system parts to be changed over time. Thus, **Designing for change PMiA14** can be seen to be a **Design decision** to facilitate **Maintaining multiple interfaces PMiA3**.

Comparing **Increasing system maintenance effort PMiA3** and **Designers minimising system change**

From the data it can be seen that the concept of ‘change’ is a maintenance cost driver; effecting change to COTS-based systems requires human effort, incurring cost. Thus, greater amounts of change can contribute to **Increasing cost**.

Therefore, the aim of **Designers minimising system change PMiA15** can be seen to be the implementation of **Design Decisions** aimed at addressing **Increasing system maintenance effort PMiA3**

Comparing **Selecting Pre-integrated components ARiB9** with **Incorporating fewer components ARiB13**

**Selecting Pre-integrated components ARiB9** can be seen as a **Design decision**, enabling system designers to implement the **DESIGN PRINCIPLE** of **Incorporating fewer components ARiB13** in a COTS-based system design, because each pre-integrated component set can be viewed as a single component unit.

Comparing **Reducing component installation effort ARiB10** with **Increasing component licensing costs ARiB12**

The choice by system developers to select pre-integrated components can be seen to be a **Design decision** aimed at contributing to **Reducing component installation effort ARiB10** at the price of **Increasing component licensing costs ARiB12**. However, designers are **Balancing cost challenges** when acknowledging that **Increasing
component licensing costs ARiB12, which can outweigh the savings in installation effort, are preferable to the increased effort of integrating and maintaining separate components.

Comparing Incorporating fewer components ARiB13 with Lowering total system costs ARiB13

The decision by system developers for Incorporating fewer components ARiB13 into COTS-based systems can be seen to be both a Design decision (which implements the DESIGN PRINCIPLE of using fewer components which are deemed beneficial to simplifying system design by reducing the number of interconnections) and a Cost reducing strategy because of the following reasons: with fewer components there can be less integration effort, less effort required to manage vendor relationships reduced exposure to forced changes to components.
Memo on Concept Design objective - PMiA2, PMiA14, ARiD23, ARiD24, ARiF1, ARiF2, ARiF3

Introduction

The Design objective of COTS-based system designers is to produce systems which solve business problems and address requirements, with reference to the availability of components. Design objectives are satisfied by designers making design decisions, normally with reference to DESIGN PRINCIPLES. However, a consequence of the practicalities of the real world can result in design objectives being addressed by design decisions leading to Conflicting design principles or by designers Balancing design principles.

Concept Design objective emerged from the following code comparisons:

Comparing Saving development time PMiA2 and Solving business problems PMiA4

Solving business problems PMiA4 expresses an objective for developing commercial software systems. In the commercial environment software systems tend to be created to solve business problems. With the focus on saving costs, Saving development time PMiA2 when producing software systems to solve business problems is a Design objective of COTS-system developers.

Comparing Identifying other cost factors ARiD23 with Statement of compatibility ARiD24

The link between codes Identifying other cost factors ARiD23 and Statement of compatibility ARiD24 is concept Design objective because, from a system design perspective, a vendor’s statement of compatibility gives system designers and developers the confidence that components will work together, minimising the effect of other cost factors.

From a cost perspective, consciously selecting components covered by a Statement of compatibility ARiD24 can also be seen to be a Cost reducing strategy employed by system designers and developers because components which have been assessed to work together are likely to cost less to integrate and maintain than components which do not have this guarantee.

Comparing Identifying other cost factors ARiD23 with Selecting certified components ARiD27

From a cost perspective, Selecting certified components ARiD27 can be seen to be related to the implementation of a Design objective because by Selecting certified components ARiD27 system designers and developers gain assurance that the components will function within specified operating environments. For example, from the interview data (ARiD) Microsoft cooperate with different vendors with the aim of developing and certifying components which will work with Windows operating systems.

Comparing Common architecture ARiF1 with Aiding component integration ARiF2
The selection of components supporting a *Common architecture ARiF1* can also be seen to be a **Design objective** of system designers who aim to select components with interfaces requiring minimal modification thus **Aiding component integration ARiF2**.

Selecting components built around a *Common architecture ARiF1* can be seen to be a **Cost Reducing Strategy** if consciously employed by system designers and developers with the aim of **Aiding component integration ARiF2** because all components supporting a common architecture will normally have compatible interfaces requiring little modification during integration.

Comparing *Common architecture ARiF1* with **Disparate vendors ARiF3**

A **Design Objective** of COTS system developers is choosing components supporting a *Common architecture ARiF1*, whether they are supplied by the same or **Disparate vendors ARiF3**, because this can contribute to reducing integration effort (effort required to modify components, change interfaces or create glue code to enable component integration).

From the data (interview ARiF) the expectation is that components supplied by the same vendor will support the same standard or architecture. However, there is no guarantee that components supplied by the same vendor will support all system requirements. Therefore, conceptually, the COTS-based design process should allow any component, irrespective of the vendor, to be integrated into a system when they provide the required functionality. However, for this to work with minimal effort the components should support the same architectural standard.
Memo on Concept Designing for change - PMiA14, PMiA15, ARiC2, ARiC7

**Introduction**
COTS-based systems can be susceptible to change factors *Beyond [the] sphere of influence* control of system developers, maintainers and administrators. For example, factors can relate to *change* requirements being ‘forced’ onto system practitioners by the following:

Component vendors: ranging from vendors recommending component upgrades and patches in order to remain in support through to vendors withdrawing components altogether forcing system practitioners to identify and implement alternative components.

System owners: ranging from system owners implementing different system requirements in order to support new business opportunities.

A consequence of *change* can also affect system stability. For example, changing COTS or other system components change can cause a system not to function as before any change activity. Furthermore, change requires testing. Problems need resolving. All of which requires human effort to resolve. Therefore, it can be seen from the data that implementing concept *Designing for change* is a requirement for COTS-based systems designers in order to deliver systems which experience minimal impact from change or have a minimal requirement for system change.

It can be seen that *Designing for change* involves designers making *design decisions* employing *DESIGN PRINCIPLES* allowing systems to be changed.

*Design for change* emerged from the following code comparisons:

Comparing *Designing for change PMiA14* and *Designers minimising system change PMiA15*

It can be seen from the data that implementing concept *Designing for change* is a requirement for COTS-based systems designers in order to deliver systems which experience minimal impact from change or have a minimal requirement for system change.

Comparing *System dependencies affecting maintenance costs ARiC1* and *Requirements changing over time ARiC4*

The assumption of *Requirements changing over time ARiC4* is that system requirements *always* change over time. A further assumption is of *System dependencies affecting maintenance costs ARiC1*. Therefore, a consequence of constantly changing system requirements on systems with greater degrees of system dependencies require more effort because system maintainers need to consider more interrelated factors.

Therefore, COTS-based system designers should employ concept *designing for change* to reduce system dependencies, enabling systems to be easily changed as a result of changing system requirements. Thus *designing for change* contributed to reducing ongoing costs.
Memo on Concept Establishing relationships - ARiD23, ARiD25, ARiD26, ARiD28, ARiL18, ARiL19, ARiL20

Introduction
From the data establishing relationships is proposed as key aim to enable system developers and maintainers to gain cooperation from other parties.

From the data the following code comparisons can be seen to link to concept establishing relationships:

Comparing Identifying other cost factors ARiD23 with Developing vendor cooperation ARiD25

The link between Identifying other cost factors ARiD23 and Developing vendor cooperation ARiD25 can be seen to be establishing relationships because an aim of system practitioners is in Developing vendor cooperation ARiD25 with the motivation of persuading component vendors to address any issues in a timely manner (such as making modifications to components etc.). It is believed that Developing vendor cooperation ARiD25 can contribute to reducing integration and maintenance costs by facilitating component modifications or other changes in a timely manner.

Comparing Developing vendor cooperation ARiD26 with Establishing vendor/client relationship ARiD28

From the data Developing vendor cooperation ARiD26 and Establishing vendor/client relationship ARiD28 can assist system developers in facilitating the resolution of technical problems and service disagreements. Thus, Developing vendor cooperation ARiD26 and Establishing vendor/client relationship ARiD28 can be seen to be contributing factors to Establishing relationships

Comparing Gaining vendor support ARiL18, Aiding integration ARiL19 and Aiding maintenance ARiL20

The link between Gaining vendor support ARiL18, Aiding integration ARiL19 and Aiding maintenance ARiL20 is concept establishing relationships because component customers establishing effective relationships with vendors can contribute to Aiding integration ARiL19 and Aiding maintenance ARiL20 as the customer and vendor are more likely to work effectively together to resolve integration and maintenance problems quickly. Furthermore, vendors are likely to benefit from additional business resulting from positive reports of good service arising from establishing good relationships.

It can be seen that establishing relationships can be measured in terms of degree (Glaser, 1978). From the data the assumption is of establishing effective relationships, which can contribute to reducing costs by Aiding integration ARiL19 and Aiding maintenance ARiL20 activities. However, at the other end of the continuum vendors or customers establishing ineffective relationships with each other can be detrimental to reducing cost as integration or maintenance problems would be less likely to be resolved effectively.

When customers are consciously establishing effective relationships with vendors, with the aim of reducing costs, it can be seen to be an example of a cost reducing strategy. It is the conscious act which defines it as a strategy (Glaser, 1978).
**Memo on Concept Implications of change - ARiC2, ARiC5**

**Introduction**

From the data **Implications of change** is associated with negative connotations for system developers and maintainers because *change* implies disruption, additional planning, testing, effort and cost. Furthermore, a consequence to changing one system part can be the requirement to change other system parts as a result of underlying dependencies.

Therefore, with reference to Glaser’s (1978; 1998) *Six C’s* theoretical coding family a *consequence* of the **Implications of change** can be *Increasing cost*.

However, with reference to Glaser’s (1978; 1998) *Six C’s* theoretical coding family it can be seen that the **Implications of change** can be measured in terms of *context*. The *process* of change can contribute to *Increasing cost* for system developers and maintainers the *result* from change activity can be beneficial to the business. For example, the resolution of software bugs or provision of additional functionality can occur as the result of *change* activity; being supplied in upgraded or patched components. New system modules created to support new business opportunities are supplied as a result of change activity.

It can also be seen that although change activity can contribute to *increasing* cost for one party, such as a system owner, it may result in additional work opportunities and revenue for other parties, such as system management companies, programmers, IT architects and other IT professionals involved with planning and managing change activity. Component vendors can also benefit from change activity as a result of charging for supplying upgraded or patched components.

From the data the *black-box* nature of COTS components does not render them immune from the effect of change. The black box concept implies that whatever occurs to a component’s inner workings the effect on the interfaces should be as described by the vendor’s documentation. However, from the data a common consequence of applying patched or upgraded components is of interfaces behaving differently from before (even when vendors have defined that the interfaces have not changed).

If the premise is that the **Implications of change** can contribute to *Increasing cost* it can be seen that *understanding* the **Implications of change** gives system developers the opportunity to plan change activity, limiting implications of change being a surprise. This can contribute to reducing cost. Thus, with reference to Glaser’s (1978; 1998) *Strategy* theoretical code a conscious decision related to *understanding Implications of change*, with the aim reducing the cost impact of change, can be seen *cost reducing strategies*.

The process of **Managing change** requires an understanding of the **Implications of change** because change can result in different implications, such as, one change requiring additional changes as a result of dependencies.

From the data the following code comparison can be seen to be linked to concept **Implications of change**
Comparing **Implications of change ARiC2** and **Underestimating COTS-based system change costs ARiC5**

From the interview data **Underestimating COTS-based system change costs ARiC5** by system practitioners is common because changes affecting COTS-based systems can not always be predicted or planned for. Furthermore, changes affecting seemingly distinct parts of COTS-based systems cannot always be performed in isolation of other work. Thus, small changes to one part of a system can require additional changes to other parts of the system. The consequence is that a seemingly small, inexpensive piece of work turns into a much larger work stream incurring much higher cost than initially envisaged. For example, changing one part of a COTS-based system can disrupt previously performed integration work necessitating this work to be redone.

Therefore, for system practitioners **Underestimating COTS-based system change costs ARiC5** can be seen to be part of the wider theme, **Implications of change ARiC2**. This statement implies that there may be other properties of the Implications of change which have not yet emerged from the data.
Memo on Concept Increasing cost - ARiB3, ARiC2, ARiC3, PMiH7, ARiK5

<table>
<thead>
<tr>
<th>Introduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>From the data concept <strong>increasing cost</strong> encompassed decisions made by system developers which result in or are believed to contribute to <em>increasing cost</em>. This is opposed to other decisions and actions which are believed to result in <em>reducing cost</em>.</td>
</tr>
</tbody>
</table>

It can be seen that **increasing cost** suggests that the broader concept **cost** can be viewed in terms of degree (Glaser, 1978; 1998). The data suggests that **cost** comprises of a continuum of factors, of which the consequences range from resulting in increasing cost through to reducing cost. It can also be assumed that some decisions would affect cost differently in terms of degree. For example, one factor may only result in a small degree of *increasing or reducing cost* whereas other factors may result in *increasing or decreasing cost* by a large amount.

From the data it is seen that system developers employ cost reducing strategies. However, there is no evidence of developers employing cost *increasing* strategies. A proposed reason for this is that for system developers representing commercial organisations the concept of reducing cost is a major driving factor because the aim of commercial organisations is to maximise profit; reducing cost can contribute to this.

Therefore, *reducing costs* can be viewed as an aim of system developers whilst *increasing costs* can be seen as an unwelcome outcome.

However, the effect of **increasing cost** can be viewed differently by different populations. For example, programmers employed to produce costly component integration code can benefit from the employment opportunity. From the system developer’s perspective this just contributes to increasing cost.

It can be seen that **Increasing cost** can be linked to concept **balancing cost challenges** in that a *consequence* of system developers and maintainers attempting to offset increasing cost factors with factors aimed at reducing costs is the outcome of balancing the cost challenges of developing and maintaining COTS-based systems.

The wider category of **controlling cost** encompasses concept **Increasing costs** because controlling cost involves all processes relating to managing cost, which are *increasing* as well as reducing cost drivers.

From the data it can be seen that a consequence of following concepts is a contribution to **Increasing cost**:

- **Increasing effort** - ARiB2, ARiB3, ARiB6, ARiB7, ARiD14, ARiD15, SDiE1, SDiE4, SDiE8, PMiG1, PMiG2, PMiG3, PMiG4, PMiG5, PMiG6, ARiL5, ARiL11.

  Human effort is related to cost as a consequence of salary costs. People cost money to employ. Furthermore, the nature of some tasks relating to building COTS-based systems requires specialist skills which can result in higher costs. Therefore, effort involving people with specialist skills would likely have a greater effect on **increasing cost** than tasks requiring people with non-specialist skills.

- **Redoing integration work** - ARiC2, ARiC9, PMiG1, PMiG2, PMiG3, PMiG4, PMiG5, PMiG6.
As consequence integrating components is Integration work, which relates to effort. It can be seen that a consequence of redoing integration work can be increasing effort because the effort used to perform the original integration work is repeated.

Staffing cost - ARiL5, ARiL6

The effect of **Staffing cost** on **increasing cost** can be measured in terms of degree. Higher staffing costs contribute more to increasing costs and visa versa.

Component licensing fees - ARiB10, ARiD34

The effect of **Component licensing fees** on **increasing cost** can be measured in terms of degree. Higher Component licensing fees contribute more to increasing costs and visa versa. From a conceptual perspective Component licensing fees encompass the intellectual property and effort performed by vendors to produce COTS components. The concept of **Component licensing fees** are a critical aspect of the concept of a COTS component. A definition of COTS components states that they are ‘sold for profit’ (Oberndorf, 2007). Implementing **Component licensing fees** is one way that vendors achieve this.

Knock-on-effect - ARiC2, ARiC10, SDiE11, SDiE12, SDiE13

A **knock-on effect** of performing some actions can be a requirement to perform additional actions as a result of dependencies, thus contributing to **Increasing cost**.

Change unpredictability - ARiC2, ARiC4, ARiC7

**Change unpredictability** can be seen as a consequence of system dependencies because the implications of changing one part of a system can result in a domino effect requiring other system parts to be changed. This series of related change requirement cannot always be predicted by system developers/maintainers when the system was originally developed. As such the result of change unpredictability can contribute to increasing cost because it is normally not planned for.

These concepts are discussed in separate memos.

From the data the following code comparison can be seen to be linked to concept **Increasing cost**

Comparing **Functional redundancy ARiB3**

A consequence of addressing **Functional redundancy ARiB3** can be **Increasing cost** for system developers because of the additional effort required to disable redundant functionality. Furthermore, tasks related to disabling functional redundancy may need to be repeated, such as following component upgrades where the upgrade process can replace the disabled functionality.

Comparing **Implications of change ARiC2** and **Knock on effect ARiC3**

The implications of change on COTS-based systems can be the ‘knock on effect’ of
further change because not all entities can be changed in isolation of other system components. The result can be **Increasing cost** for system maintainers because additional effort is required in assessing dependencies, performing additional change activity and system testing. Thus, the consequence of a seemingly small change can be a greater amount of change activity than originally envisaged because of the underlying dependencies.

Comparing **Managing change PMiH7** and **Dealing with unknown factors PMiH7**

Change and the act of **Managing change PMiH7** in COTS-based systems can be a contributing factor of **Increasing costs** as a consequence of **Dealing with unknown factors PMiH7**. The fact that the factors are considered ‘unknown’ indicates that they probably could not be predicted when a system was originally conceived. For example, when a vendor releases a component upgrade, which in turn requires an upgraded version of the operating system.

Comparing **Requiring specialist skill ARiK5**

Integration tasks (and maintenance tasks) **Requiring specialist skill ARiK5** can contribute to **increasing cost** because the cost of hiring people with these skills is normally higher than the costs required to hire normal support staff.

With reference to Glaser’s (1978) theoretical codes the relationship between **Requiring specialist skill ARiK5** and increasing cost can be explained in terms of degree. There is a cost associated with the hiring of all people. The issue here is a ‘Question of degree’ because the skills required to perform component integration and maintenance tasks tend to be more specialised and thus, incur higher cost.
Memo on Concept Increasing effort - ARiB2, ARiB3, ARiB6, ARiB7, ARiD14, ARiD15, SDiE1, SDiE4, SDiE8, PMiG1, PMiG2, PMiG3, PMiG4, PMiG5, PMiG6, ARiL5, ARiL11

Introduction
It can be seen that Increasing effort relates to the concept effort; Increasing effort reflects that effort can be measured in terms of degree. Thus, there are factors which are considered to contribute to increasing effort (of varying degrees) and factors contributing to reducing effort (of varying degrees).

However, from the data effort is not referenced to in terms of a baseline level of effort. The reference point is in terms of tasks and processes contributing to increasing or reducing effort, not a baseline level of effort to which increasing or reducing effort is measured against.

However, it can be assumed that there are baseline levels of effort for different tasks because from the interview data tasks, decisions and processes are explained with regard to their consequence on effort. Therefore, if the consequence of a course of action is increasing effort there must be point of reference level of effort to which increasing effort is assessed.

From the data concept increasing effort is seen as a major contributing factor to increasing cost. With COTS-based system development there are other significant cost contributing factors; such as the costs to purchase hardware and software licenses. However, these costs tend to be predictable and as such, can be factored in when estimating costs. An issue with the impact of increasing effort is that it cannot always be fully quantified early on in a project. The nature of the COTS-based development and maintenance process can result in tasks taking longer or unforeseen problems occurring; issues with component integration, issues following component upgrades, etc. Increasing effort to resolve problems can be the consequence. For example, on paper the integration of two components may appear straight forward. However, in practice, this integration may end up requiring the production of additional code to facilitate successful integration. Producing additional code can contribute to increasing effort.

Therefore, as a result of the unpredictable influence of effort increasing effort is perceived as having a greater impact on cost than other cost drivers. Thus, from the perspective of practitioners’ ability to influence CONTROLLING COST it can be seen that increasing effort has greater significance because of its unpredictable nature.

Furthermore, in order for commercial companies to win commissions to develop systems in they normally have to have submit bids. Bids need to reflect realistic cost estimates to realise sufficient profit margins, cover costs, as well as being pitched at a level to be favoured over other competing companies. In cases where the profit margin is tight the unpredictability of increasing effort can challenge the profitability of a winning bid.

From the data the following code comparison can be seen to be linked to concept Increasing effort:

Comparing Poor component quality ARiB2
The consequence of Poor component quality ARiB2 can be **Increasing effort** as a result of additional time and effort spent by system developers or maintainers liaising with vendors to get bugs fixed, applying patches, testing and dealing with disruptions to system operation.

Continuous Poor component quality ARiB2 can also contribute to system developers **Losing faith** with vendor’s business, especially when components continuously fail due to bugs in code.

Comparing **Functional redundancy ARiB3**

Addressing Functional redundancy ARiB3 can result in increasing costs for system developers because disabling redundant functionality requires additional effort.

Comparing **Integrating components ARiB6**

The tasks involved with Integrating components ARiB6 incur effort for system developers. Thus, it is assumed that integrating greater numbers of components contributes to **Increasing effort**.

Comparing **Increasing component integration effort ARiB7**

Increasing component integration effort ARiB7 can be seen to be a property of the concept increasing effort because increasing component integration effort is a specific example of increasing effort.

Comparing **Lacking product understanding ARiD14** with **Poor product training quality ARiD15**

The combination of the effect of Lacking product understanding ARiD14 and Poor product training quality ARiD15 can contribute to increasing cost as a result of the increasing effort incurred by system developers/maintainers in managing component problems, pursuing resolution to component problems from vendor and internal support teams and testing proposed solutions to component problems.

Comparing **Increasing number of components SDiE1, SDiE4** with **Developing additional interfaces SDiE8** and **Maintaining additional interfaces SDiE8**

The link between Increasing number of components SDiE1, SDiE4 and Developing additional interfaces SDiE8 can be viewed in terms of degree. A consequence of Developing additional interfaces SDiE8 can be additional effort. Therefore, an Increasing number of components SDiE1, SDiE4 in a system can contribute to **Increasing effort** because with more components more effort is required in Developing additional interfaces SDiE8.

Comparing **IT Architect’s perception PMiG1, Increasing cost PMiG2, Connecting more components PMiG3, Increasing effort PMiG4, PMiG5, Ongoing maintenance PMiG5, Consequence of change PMiG6**

From the data the perception of IT architects is that a consequence of including more
components in COTS-based systems is increasing effort during two phases:

1) System development: The assumption is that *Connecting more components PMiG3* can result in *increasing effort* because additional tasks needing to be performed when initially connecting interfaces of more components (From the interview there were no details on the action tasks required when integrating components, however, the assumption was that the modification required to enable two interfaces to work together requires programming effort to ensure compatibility of data being passed from one component to another – with more components the data format may have to be changed many times to facilitate data compatibility between components).

The relationship between codes when developing systems is:

The consequence of *Connecting more components PMiG3* $\rightarrow$ can be *Increasing effort PMiG4* $\rightarrow$ which contributes to *Increasing cost PMiG2*

2) System maintenance: *Ongoing maintenance PMiG5* indicates that system maintenance is not a one-time activity but involves *ongoing* tasks performed throughout the life of the system. Maintenance activity normally involves change, such as upgrading components (In such cases vendors supply new versions of components to its customers. Installing upgraded components involves change to a system because the original component is replaced).

In addition to the effort required to effect change (Installing an upgraded component requires effort in planning, assessing effect and performing the change) *increasing effort* can be required when addressing the *Consequence of change PMiG6*. For example, a *Consequence of change PMiG6* can be upgraded components working differently from original components (the nature of this difference varies – however, change can affect system stability). Therefore, addressing the consequence of change can effectively involve *redoing integration work* (such as, modifying, rewriting, testing configuration, wrapper or glue code), all requiring effort.

Therefore, if the *Consequence of change PMiG6* to one component results in *Increasing effort PMiG4* it can be assumed that the *Consequence of change PMiG6* on systems comprising of increasing numbers of components is greater levels of increasing effort increase because of the increased number of individual connections between components to be managed (*Increasing maintenance complexity*). Potentially more configuration, wrapper or glue code may need to be modified, rewritten and tested.

The relationship between codes when maintaining systems is:

The consequence of *Connecting more components PMiG3* $\rightarrow$ can lead to *Increasing maintenance complexity* $\rightarrow$ resulting in *Increasing effort PMiG4* $\rightarrow$ needed to address the *Consequence of change PMiG6* $\rightarrow$ contributing to *Increasing cost PMiG2*

Comparing *Patching ARiL10, Increasing numbers of components ARiL5, ARiL11 and Patching effort ARiL11*

The link between these codes implies that with CBD there are many tasks *Requiring effort. Patching ARiL10* of components, which normally is a requirement for
components throughout their life, is an example. Associated with patching is the effort required to test components following patching. Thus, patching does not normally occur in isolation of testing, because once a component has been patched it needs to be tested to ensure that: a) it works on its own; and b) it works with other components.

With *Increasing numbers of components* ARiL5 the expectation is that *Patching effort* ARiL11 will increase because a consequence of more components is more patching incidents, thus requiring more patching and testing effort. Therefore, *Increasing numbers of components* ARiL5, ARiL11 and *Patching effort* ARiL11 contribute to *Increasing effort*.

*Increasing numbers of components* ARiL5, ARiL11 → results in increasing *patching effort* ARiL11 → contributing to *Increasing effort*. 
Introduction

Knock-on-effect relates to the consequence of one action on another. For example, a consequence of changing one system part can be the requirement to also change other system parts as a result of dependencies. From the data knock on effect has negative connotations for system developers because it implies additional effort.

A knock-on-effect can contribute to increasing cost as the resulting actions and effort tend to be unplanned and thus not factored into original cost estimations.

Dealing with the Knock-on-effect[s] of change can form part of the process of Managing change. Managing change can involve dealing with the Knock-on-effect[s] of change because performing one change action can require additional changes, all of which require managing.

From the data the following comparison of codes are linked to concept knock-on-effect:

Comparing Implications of change ARiC2 and Replacing other components ARiC10

The Implications of change ARiC2 of COTS components can result in a Knock-on-effect of additional changes of Replacing other components ARiC10 because of underlying dependencies.

Knock-on-effect actions and changes can contribute to increasing costs for system developers/maintainers because they tend to be unplanned, as such, were not included in initial cost estimation. For example, the cost and effort estimation for upgrading one component can be increased if the task turns out to involve replacing other system components.

Knock-on-effect not only leads to Replacing other components ARiC10. The Implications of change ARiC2 on COTS-based systems can have different knock-on-effects, such as redoing integration tasks.
Memo on Concept Losing faith - ARiB2, ARiD14, ARiD17, ARiD18

**Introduction**

From the data concept **Losing faith** encompasses the lack of confidence system developers and maintainers have as a result of vendors producing poor quality products and in the inability of support organisations to provide adequate product support. However, it can be seen that **Losing faith** is not limited to organisations. **Losing faith** can occur between individuals. However, the common theme is that **Losing faith** occurs as a result of inadequate performance. The *degree* of **Losing faith** is related to the *degree* of inadequacy of performance. Thus, higher degrees of inadequacy of performance by one party can result in greater feelings of **Losing faith** in other parties.

From the data **Losing faith** can contribute to **ORGANISATIONAL ISSUES** as it relates to the inability of an *organisation* to provide quality service.

From the data the following code comparisons indicate the emergence of concept **Losing faith**:

<table>
<thead>
<tr>
<th>Comparing code: Poor component quality ARiB2</th>
</tr>
</thead>
</table>

*Poor component quality ARiB2* can contribute to **Increasing effort** as a result of system developers and maintainers spending increasing amounts of effort in liaising with vendors to get code defects fixed, applying patches, testing and dealing with disruptions to system operation.

Continuous *Poor component quality ARiB2* can also contribute to system developers **Losing faith** with vendor’s ability to provide quality products, especially when components continuously fail due to defects.

Comparing **Lacking product understanding ARiD14** with **Differing product support arrangements ARiD17, ARiD18**

It can be seen that a consequence of **Lacking product understanding ARiD14** and **Differing product support arrangements ARiD17, ARiD18** can result in system developers and maintainers **Losing faith** with vendor’s and internal support team’s ability to support products.
Introduction

Maintenance complexity conceptualises the number of interrelated tasks, relationship and dependencies between tasks, which have to be performed when maintaining COTS-based systems. For the purpose of this study, maintenance encompasses tasks occurring after system implementation.

With reference to Glaser’s (1978; 1998) theoretical coding families Maintenance complexity can be measured in terms of degree. For example, maintenance tasks involving changing single components with no underlying dependencies can be seen as possessing lower degrees of Maintenance complexity. Conversely, maintenance activity involving many components, with higher degrees of dependencies, requiring people with specialist skills can be seen to be associated with higher degrees of Maintenance complexity.

With reference to Glaser’s (1978; 1998) Six C’s theoretical coding family the expectation is that Maintenance complexity is influenced as a result of System complexity, rather than the other way round. Thus, the consequence of a higher degree of system complexity is likely to be an increasing degree of Maintenance complexity.

From the interview data Maintenance complexity emerged from the following code comparisons:

Comparing Maintaining multiple interfaces PMiA3 and System dependencies PMiA19

From the data the relationship between Maintaining multiple interfaces PMiA3 and System dependencies PMiA19 can be seen to contribute to increasing maintenance complexity because a consequence of ‘increasing’ System dependencies PMiA19, combined with Maintaining multiple interfaces PMiA3, can result in many factors for system maintainers to consider. Furthermore, there is a greater likelihood for problems occurring as a result of maintainers failing to appreciate the complexities and dependencies within systems.

Comparing Multiple vendors PMiA10 and Conflicting maintenance schedules PMiA10

A result of selecting components from Multiple vendors PMiA10 can be Conflicting maintenance schedules PMiA10 which can contribute to increasing maintenance complexity because of the number of factors which need to be considered when planning and performing system maintenance tasks.

Comparing Disparate vendors ARiF3 with Underlying dependencies ARiF4

A consequence of combining Disparate vendors ARiF3 with Underlying dependencies ARiF4 can be seen to contribute to Maintenance complexity because there can be varying Underlying dependencies ARiF4 arising from components
supplied by Disparate vendors ARiF3 adding to the complexity of maintaining the systems. For example, upgrading one vendor’s component may require other components to be upgraded in order for stability to be maintained. Therefore, maintenance complexity is defined in terms of the number of factors which have to be considered during the life of a system. The number of different vendor’s products involved in a system can contribute to maintenance complexity.

Comparing Underlying dependencies ARiF4 with Managing multiple vendors components ARiF7

The combination of codes Underlying dependencies ARiF4 with Managing multiple vendors’ components ARiF7 can contribute to increasing Maintenance complexity for system maintainers because of two factors: firstly, when all vendors do not implement the same component upgrade cycle additional planning effort can be required by system maintainers to schedule all component upgrades. Secondly, with underlying dependencies between different vendor’s components, such as the requirement for other components to be at specific release levels or specific operating system versions, then Maintenance complexity can increase as a consequence of the number of factors be considered and tasks to be performed. For example, an operating system upgrade may be a pre-requisite of upgrading a component.

Comparing Underlying dependencies ARiF4 with Appreciating hidden dependencies between components ARiF8

Appreciating hidden dependencies between components ARiF8 in order to understand the nature of Underlying dependencies ARiF4 between components can contribute to reducing Maintenance complexity because it can help system developers to better plan the maintenance tasks involved in maintaining COTS-based systems with minimal disruption to system stability.

There is the question of when Appreciating hidden dependencies between components ARiF8 should occur. If this is achieved before system development time system developers may be in a position to choose alternative components in order to reduce the degree of Underlying dependencies ARiF4 between components, thus contributing to reducing Maintenance complexity. However, Appreciating hidden dependencies between components ARiF8 once a system has been implemented may have little effect in reducing Maintenance complexity because any Underlying dependencies ARiF4 already exist.

Comparing IT Architect’s perception PMiG1, Increasing cost PMiG2, Connecting more components PMiG3, Increasing effort PMiG4, PMiG5, Ongoing maintenance PMiG5 and Consequence of change PMiG6

From the data the perception of IT architects is that a consequence of including more components in COTS-based systems is increasing effort during two phases:

1) System development: The assumption is that Connecting more components PMiG3 can result in Increasing effort because of additional tasks needing to be performed when initially connecting interfaces of more components (From the interview there were no details on the action tasks required when integrating components, however, the assumption was that the modification required to enable two interfaces to work
together requires programming effort to ensure compatibility of data being passed from one component to another – with more components the data format may have to be changed many times to facilitate data compatibility between components).

The relationship between codes when developing systems is:

The consequence of Connecting more components PMiG3 can be Increasing effort PMiG4 which contributes to Increasing cost PMiG2

2) System maintenance: Ongoing maintenance PMiG5 indicates that system maintenance is not a one-time activity but involves ongoing tasks performed throughout the life of the system. Maintenance activity normally involves change, such as upgrading components (In such cases vendors supply new versions of components to its customers. Installing upgraded components involves change to a system because the original component is replaced).

In addition to the effort required to effect change (Installing an upgraded component requires effort in planning, assessing effect and performing the change) Increasing effort can be required when addressing the Consequence of change PMiG6. For example, a Consequence of change PMiG6 can be upgraded components working differently from original components (the nature of this difference varies – however, change can affect system stability). Therefore, addressing the consequence of change can effectively involve Redoing integration work (such as, modifying, rewriting, testing configuration, wrapper or glue code), all requiring effort.

Therefore, if the Consequence of change PMiG6 to one component results in Increasing effort PMiG4 it can be assumed that as the number of system components increases the Consequence of change PMiG6 can contribute further to increasing effort as a result of the greater complexity of connections between components to be managed. This can contribute to increasing Maintenance complexity because more configuration, wrapper or glue code may need to be modified, rewritten and tested.

The relationship between codes when maintaining systems is:

The consequence of Connecting more components PMiG3 can contribute to increasing maintenance complexity resulting in Increasing effort PMiG4 needed to address the Consequence of change PMiG6 contributing to Increasing cost PMiG2

Comparing Maintaining multiple interfaces PMiA3 with Selecting architecturally incompatible components PMiA8
Multiple interfaces, resulting from the integration of many components, can add complexity to system architecture because many factors have to be considered when developing a system. System complexity can also be compounded if the components making up the system do not support the same architectural standards, which can occur when components are supplied by different vendors.

Maintaining multiple interfaces PMiA3 can be defined in terms of degree because multiple varies in number from two through to infinity. However, the assumed association is that multiple interfaces can increase as a result of more interfaces associated with each component or as a result of greater numbers of single-interface
components.

Interfaces tend to require maintaining over time. An assumption is that maintenance complexity can change in relation to the number of connected interfaces. Thus, as the number of connected interfaces increases the greater its effect on maintenance complexity because more factors need to be considered, dependencies assessed etc.

The consequence of designers Selecting architecturally incompatible components PMiA8 on Maintaining multiple interfaces PMiA3 is to contribute further to maintenance complexity because there are even more factors related to sustaining system stability.

Therefore, the association between Maintaining multiple interfaces PMiA3 and Selecting architecturally incompatible components PMiA8 is to contribute to Maintenance complexity

Comparing Maintaining multiple interfaces PMiA3, Multiple vendors PMiA10 and Conflicting maintenance schedules PMiA10

From the data the assumption is that the costs of Maintaining multiple interfaces PMiA3 supplied by Multiple vendors PMiA10 will be higher, compared with the maintenance costs of components supplied by the same vendor, because the components may not adhere to the same architectural standard.

Additionally, the assumption behind codes Maintaining multiple interfaces PMiA3 and Conflicting maintenance schedules PMiA10 is that components supplied by different vendors will not have the same maintenance schedule, thus adding complexity to system maintenance because different maintenance tasks may have to be performed at different times.

Therefore, Maintaining multiple interfaces PMiA3 of components supplied by Multiple vendors PMiA10 which have Conflicting maintenance schedules PMiA10 can contribute to Maintenance complexity.

Comparing code: Synchronising maintenance schedules ARiD35

Synchronising maintenance schedules ARiD35 contributes to Maintenance complexity because when a vendor’s upgrade policy does not match their customers’ organisational maintenance policies more factors have to be considered by system maintainers when managing ongoing system maintenance to ensure system stability.
Memo on Concept Managing change - PMiH2, PMiH1, PMiH7, ARiK8

Introduction

Managing change conceptualises the processes required to plan and implement change activity.

From the data Managing change emerged from the following code comparisons:

Comparing Changing over time PMiH2 and Legacy implementation PMiH1

From the data a certainty is of Legacy implementation[s] PMiH1 and other entities Changing over time PMiH2. Software systems can change over time, Operating systems require upgrading, COTS-components evolve over time or are withdrawn, business requirements change and new business opportunities arise. Therefore, Managing change is required by system developers and maintainers to react to the requirement of Changing over time PMiH2 in order to keep systems functioning.

Managing change is a contributing factor of CONTROLLING COST because it involves effort and in many cases, specialist skills. Thus, in order to successfully control costs managing change should be performed effectively.

A consequence of Designing for change can be to make Managing change easier for system developers and maintainers as a result of applying design principles allowing for parts of COTS-based systems to be changed with minimal impact the rest of the system. However, Designing for change normally has to be thought of early on in the design process because it may not be possible to retrofit this design principle after system implementation.

Thus, with reference to Glaser’s (1978; 1998) theoretical codes a consequence of implementing a designing for change policy is to make managing change easier (costing less, less onerous) for system maintainers by allowing system parts to be changed with little effect on the rest of the system.

The absence of Designing for change can result in changing one system part leading to a Knock-on effect of the requirement to change other system parts.

Comparing Managing change PMiH7 and Dealing with unknown factors PMiH7

A consequence of changing COTS-based systems and the costs of Managing change PMiH7 are contributing factors to Increasing costs because of the effect of dealing with unknown factors, which may not be predicted when systems are originally conceived. For example, when vendors releases upgraded components requiring an upgraded version of the operating system.
Memo on Concept Organisational concerns - ARiB2, ARiB15, ARiB16, ARiB17

Introduction

Organisational concerns encapsulate the confidence and concern one party has over another organisation’s ability to provide a service. This can be defined in terms of degree. Thus, the lower the confidence of an organisation the higher the organisational concerns.

It can also be seen that the degree of Organisational concerns can be affected by the degree issues of concern are Beyond [the] sphere of influence. Thus it is likely that one party would express a higher degree of organisational concern when the issues of concern were completely beyond their sphere of influence. For example, in cases where vendors provide poor quality components, are unwilling to cooperate with system developers to improve product quality and where the choice of components are beyond the sphere of influence of system developers because no alternative options are available it could be assumed that system developer’s organisation concerns over a vendor’s organisation could be higher than if he/she had the option to source alternative components from different vendors.

Concept Organisational concerns emerged from the comparison of the following codes:

Comparing Poor component quality ARiB2 and Vendor’s organisational complexity ARiB15

A consequence of a combination of Poor component quality ARiB2 and Vendor’s organisational complexity ARiB15 can be Organisational concerns of the vendor’s business by system developers and maintainers. This is because the complexity of a vendors’ organisation can affect the support quality provided for their components. Thus, the consequence of a higher degree of Vendor’s organisational complexity ARiB15 can be Poor component quality ARiB2. From the interview data it was also observed that Poor component quality ARiB2 tended to emanate from larger vendor organisations. The perception was that there was less synergy and cooperation between departments of larger organisations.

Comparing Poor component quality ARiB2 and Poor vendor support ARiB16, ARiB17

A consequence of Poor vendor support ARiB16, ARiB17 can be Poor component quality ARiB2 arising from vendor organisations where there appears little synergy, communication and cooperation between internal departments to fix bugs and resolve other issues. This effect can be made worse when components span several vendor departments. Therefore, after experiencing a history of Poor component quality ARiB2 combined with Poor vendor support ARiB16, ARiB17 system developers and maintainers express Organisational concerns with a vendor’s business.
Memo on Concept Redoing integration work - ARiC2, ARiC9, PMiG1, PMiG2, PMiG3, PMiG4, PMiG5, PMiG6

Introduction

Concept **Redoing integration work** is a property of concept ‘redoing’ – in this case *redoing* integration work. *Redoing* implies that an action or task has already been performed, thus, resulting in it being performed again. Therefore, when a task or action has incurred effort it can be seen that redoing the task contributes to increasing effort.

From the data integration work equates to integration effort.

From the data **Redoing integration work** can be seen as a consequence of change, where change activity can overwrite previously performed integration work necessitating the consequence of **Redoing integration work**.

Redoing integration work relates to the category of **CONTROLLING COST** because it is a contributing factor of increasing effort.

From the data the following comparison of codes are linked to concept **Redoing integration work**:

Comparing **Implications of change ARiC2** and **Redoing integration work ARiC9**

It can be seen from the data that a consequence of change to COTS-based systems can result in system developers/maintainers **Redoing integration work**. For example, a result of changing COTS components, such as upgrading a component, can necessitate the rewriting of integration code when the vendor has changed the way the upgraded component works. **Redoing integration work** therefore adds cost for COTS-based system maintainers because of the human effort required in producing integration code.

Comparing **IT Architect’s perception PMiG1, Increasing cost PMiG2, Connecting more components PMiG3, Increasing effort PMiG4, PMiG5, Ongoing maintenance PMiG5, Consequence of change PMiG6**

From the data a perception of IT architects is that a consequence building COTS-based systems from more components can be **increasing effort** during two system lifecycle phases:

1) System development: The assumption is that **Connecting more components PMiG3** can result in **increasing effort** because additional tasks needing to be performed when initially connecting interfaces of more components (From PMiG interview there were no details on the actual tasks involved with integrating components. However, the assumption was that component integration requires programming effort to facilitate compatibility of data being passed between components. With additional numbers of components the data format may have to be changed frequently to enable data compatibility between components).

The relationship between codes when developing systems is:

The consequence of **Connecting more components PMiG3** → can be **Increasing effort**
2) System maintenance: *Ongoing maintenance PMiG5* indicates that system maintenance is not a one-time activity but involves *ongoing* tasks performed throughout the life of the system. Maintenance activity normally involves *change*, such as upgrading components (in such cases vendors supply new versions of components to its customers. Installing upgraded components involves change to a system because the original component is replaced).

In addition to the **effort** required to effect changes (for example, upgrading components tends to require effort in planning, assessing consequences and performing change) **increasing effort** can be required when addressing the *Consequence of change PMiG6*. Upgraded components working differently from original components can be an example of *Consequence of change PMiG6* (the nature of components working differently can vary) Therefore, addressing the *Consequence of change PMiG6* can necessitate **redoing integration work** (such as, modifying, rewriting, testing configuration, wrapper or glue code), contributing to increasing effort.

Therefore, when the *Consequence of change PMiG6* to one component is *Increasing effort PMiG4* it can be assumed that the *Consequence of change PMiG6* on systems comprising of greater numbers of components can be measured in terms of degree. The consequence of changing more components can have a greater contribution to increasing effort because of the number of individual connections between components to be managed (*Increasing maintenance complexity*). Potentially more configuration, wrapper or glue code may need to be modified, rewritten and tested.
Memo on Concept Reducing integration effort - PMiA2, PMiA8, ARiD1, ARiD2, ARiD3, ARiD4, ARiF1, ARiF2

**Introduction**
Integration effort relates to the human effort required to integrate COTS components. From the data it can be seen that concept integration effort is a contributing factor of cost because human effort incurs financial cost related to the skill and salary to perform these tasks.

It can be seen that integration effort is a factor which can vary as a consequence of other actions. Thus, the resulting amount of integration effort relates to the types of tasks to be performed, the skill required to perform tasks, the amount of time required to perform tasks.

Integration Action/decision → can result in → Increasing or Reducing integration effort

However, from the data there was no indication of what constituted benchmark integration effort. For example, if certain decisions are deemed to contribute to reducing integration effort the baseline amount of integration effort to which this is compared against was not established. The data only indicates that there are decisions, actions etc. which can result in increasing integration effort and decisions which can result in reducing integration effort.

Therefore, Integration Effort can be viewed as a continuum, of which Reducing integration effort is at one end.

Increasing integration effort ← Integration effort → Reducing integration effort

Thus, the data indicates that actions, principles and decisions are endorsed by COTS system developers because they are believed to contribute to Reducing integration effort.

From the data the following comparison of codes are linked to concept Reducing integration effort:

Comparing **Saving development time PMiA2** and **Integration effort PMiA8**

It can be seen from the data that there is a relationship between codes *System developers Saving development time PMiA2* and *Integration effort PMiA8* because the amount of integration effort can affect development time; the expectation is more integration effort will result in longer development time. Therefore, less integration effort should contribute to reducing system development time.

For the purposes of this study (and from the data) ‘integration effort’ is taken to mean ‘enabling one or more system components (COTS components or other system parts) to communicate together to form a software system’.

Comparing **Sourcing components from same vendor ARiD1** with **Vendor testing integration ARiD2**, **Saving time ARiD3**, and **Saving testing effort ARiD4**

Codes **Sourcing components from same vendor ARiD1** and **Vendor testing integration ARiD2**, **Saving time ARiD3**, and **Saving testing effort ARiD4**
ARiD2, Saving time ARiD3, and Saving testing effort ARiD4 are linked to the concept of reducing integration effort for system developers because it is assumed that vendors have performed integration testing ensuring that their set of components integrate together with minimal effort and time. The assumed consequence of Sourcing components from same vendor ARiD1 is that this saves system developers the time and effort of performing some component integration testing tasks – they just need to test that the components work within the system. Therefore, if consciously chosen, Sourcing components from same vendor ARiD1 is a strategy aimed at reducing integration effort.

Comparing Common architecture ARiF1 with Aiding component integration ARiF2 and Reducing integration costs ARiF2

Selecting components built around a Common architecture ARiF1 can be a Cost Reducing Strategy employed by system developers for Aiding component integration ARiF2 because it is assumed that each component has compatible interfaces requiring little modification.

The selection of components supporting a common architecture can also be seen to be a design objective of system architects who aim to select components with interfaces requiring minimal modification thus Aiding component integration ARiF2, reducing integration effort and Reducing integration costs ARiF2.
Memo on Concept Reducing potential problems - PMiH6

Introduction
From the data COTS-based system developers and maintainers understand that certain design decisions can increase the chance for potential problems. Resolving problems can contribute to increasing costs as a result of the additional time, effort and resources required to perform problem determination and resolution tasks. Therefore, a conscious decision aimed at reducing potential problems can be viewed as a cost reducing strategy when made in relation to reducing the costs associated with dealing with problems. Choices which merely result in reducing potential problems should be seen as a consequence of that choice, rather than as part of a strategy.

From the data the following code comparison is linked to concept Reducing integration effort:

Comparing Reducing system maintenance costs PMiH6 and Reducing potential problems PMiH6

The conscious choice of building COTS-based systems from fewer components is believed to contribute to Reducing system maintenance costs PMiH6 by Reducing potential problems PMiH6 resulting from integrating fewer interfaces and lessening the number of vendor interactions and associations.

Therefore, a design decision of building systems using fewer components can be seen as a cost reducing strategy when consciously employed by system developers because it is believed to contribute to reducing ongoing costs.
Memo on Concept Reducing user intervention - ARiB9, ARiB10, ARiC15, ARiC17

Introduction
Concept Reducing Human Intervention relates to the concept of Human intervention. There is a relationship between Human intervention and effort, and thus, cost because Human intervention implies someone performing an action: human effort, which can incur cost.

With regard to the development and maintenance of COTS-based systems tasks involving human intervention incur cost. Tasks requiring specialist skill normally incur greater cost.

Therefore, actions resulting in reducing human intervention can contribute to reducing effort and thus, reducing cost. With reference to Glaser’s (1978) theoretical codes Reducing human intervention can be seen as a cost reducing strategy if the underpinning decisions were made with the aim of consciously reducing human intervention. Where this is not the case then reducing human intervention merely becomes a consequence of an action.

Reducing human intervention can also be seen to relate to the ‘degree’ coding family (Glaser, 1978) because it can be viewed as a continuum ranging from reducing human intervention by a small amount through to reducing human intervention completely. However, reducing human intervention implies that there is the potential for an action to occur without human intervention. Some tasks may not have this potential.

From the data the following comparison of codes resulted in the link to Reducing human intervention:

Comparing Selecting Pre-integrated components ARiB9 with Reducing component installation effort ARiB10

Selecting Pre-integrated components ARiB9 can contribute to Reducing component installation effort ARiB10 for system developers by Reducing human intervention (However, it should be noted that with pre-integrated components the component integration effort is performed by the vendor).

Comparing Reducing ongoing costs ARiC15 with Choosing pre-integrated components ARiC17 07FEB09

Choosing pre-integrated components ARiC17 can contribute to Reducing ongoing costs ARiC15 by Reducing human intervention, thus reducing human effort. System developers consciously Choosing pre-integrated components ARiC17 can also be seen as a Cost reducing strategy employed by system developers by reducing human effort to integrate and maintain a set of components.
Introduction

**Relationship complexity** conceptualises the complexities between human relationships, as opposed to technical complexities encompassed in *system* or *maintenance* complexity. Contributing factors to **Relationship complexity** can be the number of vendors and support organisations system developers and maintainers have to deal with. Therefore, systems comprising of components supplied from different vendors can be associated with higher degrees of support complexity for system maintainers as a result of the number of different support organisations and support personnel which have to be dealt with in order to receive support for the system as a whole.

The ability to deal with **Relationship complexity**, throughout a system’s lifespan, can be a factor of MANAGING COMPLEXITY as a result of system developers and maintainers having to deal with and negotiate with different populations, including vendors, support teams and customers.

With reference to Glaser’s (1978; 1998) theoretical codes **Relationship complexity** can be defined in terms of *degree*, ranging from lower through to higher degrees of **Relationship complexity**. It can be seen that higher degrees of **Relationship complexity** can contribute further to MANAGING COMPLEXITY as a result of greater numbers of populations to be dealt with.

There can be an inverse relationship between **Relationship complexity** and **Support quality** because as the complexity of support relationships increase the quality of support provided can decrease as a result of greater numbers of people to be dealt with.

It can be seen that **Denying responsibility** can compound **Relationship complexity**. For example, in situations where system developers or maintainers are required to deal with different support organisations for problem resolution the consequence of vendors denying responsibility for problem root cause can add to **Relationship complexity** and contribute further to the effort required in MANAGING COMPLEXITY.

**Relationship complexity** emerged from the following code comparisons:

Comparing Increasing number of components SDiE1, SDiE4 with Isolating problems SDiE9

From a support perspective, the number of components in a system affects the ability of system developers/maintainers and vendors to isolate problems.

When there is an interaction of different vendors’ products, in-house built glue code, wrapper code and possibly legacy application code Isolating problems SDiE9 can be challenging for customers (system developers/maintainers) and vendors. This can be compounded with systems containing more components because there may be more coding factors, such as interfaces, to consider as well as more customer/vendor
relationships to manage. All of which can contribute to Relationship complexity as a result of having to deal with greater numbers of vendors’ support groups or different support teams within the same vendor organisation.

From the data the link between concept Relationship complexity and category MANAGING COMPLEXITY appears causal-consequence and degree because, for example, a consequence of higher degrees of Relationship complexity, resulting from dealing with greater numbers of vendor support groups, contribute to more factors to consider when MANAGING COMPLEXITY, as a result of the number of different people system developers and maintainers have to liaise with when isolating problems.

Comparing Increasing number of components SDiE1, SDiE4 with Vendor homogeneity SDiE5

From the data code Vendor homogeneity SDiE5 encapsulates the notion of different components being supplied by the same vendor. From the data (previous interviews PMiA and ARiB) the choice of system developers to build COTS-based systems from components supplied by the same vendor was considered to be a design decision contributing to reducing costs because of the assumption that the components would be designed to integrate together. Thus, from a design perspective sourcing components from the same vendor is seen to reduce the effort required to tailor and integrate components supplied by different vendors. A further assumption from the interview data was that the maintenance effort would be less for components supplied by the same vendor because vendor would test and verify the integration of upgraded components with all of their products.

Sourcing components from the same vendor (Vendor homogeneity SDiE5) was also deemed to reduce the development and maintenance costs when building systems from greater numbers of components because of the assumption that components supplied by the same vendor are designed to integrate together with minimal effort.

However, from this data (Interview SDiE) large vendors producing many COTS components may comprise of a more complex internal organisational structure, whereby different components are developed and supported by different internal departments or teams. From a product support perspective the different departments of some large vendor organisations can appear to system developers/maintainers as virtually different companies, with little synergy and cooperation between departments. Thus, from a system developer’s perspective the cost benefits of sourcing components from some large vendors can be challenged as a result of a contribution to increasing Relationship complexity manifested by the requirement of dealing with different support groups when requiring product support.

Another link between codes Increasing number of components SDiE1, SDiE4 and Vendor homogeneity SDiE5 can be concept balancing cost challenges because, on the one hand, building systems from more components can add cost for system developers due to the increasing cost of developing and maintaining more interfaces. Conversely, the cost of using a greater number of components can be offset by sourcing them from the same vendor because the assumption is that components supplied by the same vendor will integrate together with minimal effort. However, the cost benefit of sourcing the components could be diminished if the vendor’s internal
support structure makes it difficult to gain adequate support for the variety of products.

Comparing Vendor homogeneity ARiK9, Quality of support ARiK10 and Sourcing components from one vendor ARiK11

Codes Vendor homogeneity ARiK9, Quality of support ARiK10 and Sourcing components from one vendor ARiK11 can be seen to be linked to the concepts of Support quality and Relationship complexity. Vendor homogeneity ARiK9 – the number of vendors contributing to supplying components to a system can be seen to have a bearing on the functional compatibility of components because the assumption is that components supplied by the same vendor are more likely to be compatible with each other. However, from system developers’ perspective, the general assumption is that vendor homogeneity affects Relationship complexity because as the number of vendors supplying components to one system increases more support teams may have to be contacted when dealing with problems, product upgrades, etc. Thus, a contributing factor to Relationship complexity can be the number of different support teams.

The Quality of support ARiK10 provided by individual support teams may be high but a system developer’s perception of having to deal with increasing numbers of support teams, as the number of components being supplied by different vendors increases, can collectively add to feeling of diminishing support quality as a result of increasing Relationship complexity.

With reference to Glaser’s (1978) theoretical coding families Relationship complexity can be defined in terms of ‘degree’, related to the size of vendor organisations. For example, an assumption is that acquiring components from one vendor results in a system developers/maintainer being more likely to deal with one support team. At the ‘smaller vendor size’ end of the continuum this may be true as one team may support all of the vendor’s products. In this case Support quality can also be higher as a result of one support team providing assistance for all products (however, it should be stated that a single support team can still provide poor quality service). Conversely, in this scenario Relationship complexity is likely to be low as a as result of system developers/maintainers having to only contact one support team for all product enquiries.

However, at the other end of the ‘vendor size’ continuum, as vendor organisations get larger each product may be supported be different support teams or departments within the same organisation. In such cases Support quality can decrease when each support team/department can appear as separate organisations, with little synergy between each team/department (see Interview SDiE xxxx xxxx). Relationship complexity may increase as a result of system developers/maintainers having to contact more than one support team or department to receive support for different components supplied by the same vendor.

Attributing accountability

Introduction
In complex environments the root cause of problems may not be obvious. This can be further compounded when vendors deny responsibility for problems because they feel the cause is beyond the scope of their code [normally black box code]. Comparing
complex systems comprising of more components, containing greater numbers of interfaces, glue code, wrapper code, **Attributing accountability** may be one challenge of isolating problems because vendors may be unable to locate the root cause of a problem or unwilling to cooperate with other vendors (or other departments within the same vendor) to aid problem isolation. It could be suggested that **Attributing accountability** of problems in systems with fewer components and integrated parts is more straightforward because there are less factors to consider.

**Attributing accountability** emerged from the following code comparisons:

However, from the interview data it is not just an *Increasing number of components* SDiE1, SDiE4, linked to *Isolating problems* SDiE9, which affects **Attributing accountability**, but the number of involved parties and complexities of the relationships between system developers and maintainers and vendors which may be important. For example, a system containing a greater number of components, possibly supplied by the same vendor, and where a good working relationship exists between customer and vendor and where the vendor willingly cooperates with system developers/maintainers to aid problem resolution the challenge of **Attributing accountability** by system developers/maintainers may not exist because the vendor assumes a degree of responsibility.

Therefore, it can be seen that the association between codes *Increasing number of components* SDiE1, SDiE4 and *Isolating problems* SDiE9 and concept **Attributing accountability** is a contributing factor to category **MANAGING COMPLEXITY** as a result of the *degree* of **Relationship complexity**.

The affect of reducing **Relationship complexity**, such as where system developers/maintainers only have to deal with one vendor, **Attributing accountability** may be more willingly agreed compared with cases where system developers/maintainers are dealing with multiple vendors or multiple vendor departments.
Memo on Concept Resisting change - PMiH11, PMiH13, PMiH16

Introduction

From the data, the concept emerged as a consequence of people Resisting change to adopting the COTS-based approach. The culture within some organisations is to accept change with little resistance. However, in other organisations when staff members have been used to specific ways of working system developers can encounter resistance to change when implementing different systems. Therefore, with reference to Glaser’s (1978; 1998) theoretical codes it can be seen that Resisting change can be defined in terms of degree, ranging from Resisting change to a small degree, through to Resisting change by larger degrees.

It can be seen that Resisting change is not only related to the domain of COTS-based development but can occur in any area where people do not willingly embrace change.

Resisting change can be seen to be an ORGANISATIONAL ISSUE because it relates to organisational cultures of resisting change.

Resisting change can contribute to Increasing cost as a consequence of additional time and effort required to persuade people to adopt different working practices.

Component Resisting change emerged from the following code comparisons:

Comparing Resisting business process change PMiH11, PMiH16 and Attempting to consolidate business processes PMiH13

From the data Resisting business process change PMiH11, PMiH16 by organisations can be a challenge for the use of the COTS-based approach. Generally, organisations have business processes. The COTS-based approach can result in a need for business processes to change in order to match the way the COTS components work. Attempting to consolidate business processes PMiH13 can result in organisations Resisting change.

Furthermore, the business processes within organisations can differ across internal departments. However, the use of the COTS-based approach can require a company to standardise processes across departments, which can lead to personnel resisting change. Producing custom processes for different departments can be a solution, but this can be difficult and add cost as due to the effort required to tailor COTS components.
Memo on Concept Support quality - ARiK9, ARiK10, ARiK11, ARiL16 –
26JULY09

Introduction
Support quality conceptualises the quality of support of components provided by one organisation to another. Support quality can be defined in terms of a continuum (Glaser, 1978; 1998), ranging from values, such as, poor through to excellent Support quality.

From the data Support quality was defined in conjunction with concept Support complexity. From a support perspective as the number of components supplied by different vendors increases support quality can decrease and support complexity can increase. This is because more support teams have to be contacted when requiring support. Thus, support complexity can be defined in terms of number of different support teams.

The support quality of some vendors support teams may be higher when considered in isolation. However, system developers and maintainers’ perception of having to deal with increasing numbers of support teams, as the number of components being supplied by different vendors increases, can contribute to a feeling of diminishing support quality (a single point of contact for gaining support for products supplied from multiple vendors could enhance support quality).

With reference to Glaser’s (1978; 1998) theoretical coding families Support quality and support complexity can be defined in terms of ‘degree’, measured by the size of vendor’s organisation. For example, the assumption from system developers is that acquiring components from one vendor customers are more likely to deal with one support team. At the ‘smaller vendor size’ end of the continuum this may be true as one team may support all products. In this case support quality tends to be higher as one support team can provide assistance for all products (however, it should be stated that a single support team can still provide poor quality service). Support complexity can be seen to be low when required to contact only one support team for all product enquiries.

However, at the other end of the ‘vendor size’ continuum, as vendor organisations get larger each product may be supported be different support teams or departments within the same organisation. In such cases support quality can decrease because each support team/department can appear as separate organisations, with little synergy between each team or department. Support complexity may increase as a consequence the need to contact more than one support team or department in order to receive support for different components supplied by the same vendor.

It can be seen that the perceived level of Support quality one party provides to others can affect their perceived levels of Organisational concern. Thus, poor Support quality may contribute to a higher degree of Organisation concern of the organisation delivering poor quality support. Furthermore, Support quality may be Beyond [the] sphere of influence of one organisation when the support service is provided by another organisation.

From the data Support quality emerged from the following code comparisons:

Comparing Vendor homogeneity ARiK9, Quality of support ARiK10 and
Comparing Codes Vendor homogeneity ARiK9, Quality of support ARiK10 and Sourcing components from one vendor ARiK11 can be seen to be related to the concepts of support quality and support complexity. Vendor homogeneity ARiK – the number of vendors contributing to supplying components to a system can be seen to have a bearing on the functional compatibility of components. However, from system developers’ perspective, the general assumption is that vendor homogeneity affects the support complexity because as the number of vendors supplying components increases more support teams have to be contacted when dealing with problems, product upgrades, etc. Thus, support complexity can be defined in terms of number of different support teams.

Comparing code: Poor vendor support ARiL16

Poor vendor support ARiL16 demonstrates a property of concept Support quality which can be measured in terms of a continuum ranging from poor support quality through to good support quality.

From the data Support quality was defined in terms of a vendor’s lack of ability supporting their components. The vendor had good intentions to provide good quality support but was lacking ability to achieve this.

However, it can be seen that lack of integrity, care etc. could also result in poor quality, in which case the vendor does not have good intentions of providing good quality support.
Memo on Concept System complexity - ARiB3, ARiC15, ARiC16, ARiD1, ARiD9, PMiA3, ARiL22, ARiL21

Introduction

System complexity conceptualises the number of interrelated factors making up a system. It can be seen that System complexity can be applied to any system where interrelating factors occur. However, for the purposes of this study System complexity relates to COTS-based systems.

With reference to Glaser’s (1978; 1998) theoretical coding family, the Degree Family, System complexity can be seen in terms of a continuum, ranging from a low to high degrees of system complexity. Furthermore, there are factors and processes which contribute to reducing or increasing System complexity.

From the interview data it can be seen that System complexity is related to different factors:

The number of different connections. Thus, systems containing greater numbers of components and interconnected interfaces can contribute to increasing System complexity.

The number of dependencies. A consequence of systems with more dependencies can be defined in terms of higher degrees of System complexity because changing one system part can require further changes to other system parts. Therefore, the knock-on effect of a seemingly small change can be higher levels of change activity.

Amount of functionality. Components providing greater amounts of functionality can contribute to increasing complexity, compared with components providing simple instances of functionality. Thus, the cumulative affect of combining components of increasing complexity can contribute to increasing System complexity.

As with cost, examples from the data indicate factors relating to either increasing or reducing System complexity. However, there is no indication of a base lined level of system complexity to which increasing or reducing should be compared against.

It can be seen that that System complexity can relate to category MANAGING COMPLEXITY because system designers, developers and maintainers endeavour to manage system complexity when designing, developing or maintaining COTS-based systems. From the data in can be seen that complexity is a contributing factor of cost. The assumption is that a consequence of increasing complexity is increasing cost because of greater numbers of factors needing to be considered, requiring more time and higher levels of skill. Thus, MANAGING COMPLEXITY can involve decisions and actions with the aim of reducing complexity as a means of reducing costs.

From the data System complexity emerged from the following code comparisons:

Comparing code: Selecting Pre-integrated components ARiB9

Selecting Pre-integrated components ARiB9 can contribute to reducing System complexity because when vendors control the integration and ongoing maintenance
of a pre-integrated set of components (a pre-integrated component collection can be viewed as a single entity – system complexity can be measured by the number of separate connections managed by system designers, developers and maintainers) system designers and developers only have to be concerned with the integration of this entity with remaining system parts. The consequence is to reduce the number of component connections requiring management.

Comparing Selecting fewer components ARiC15 and Choosing components supplied by same vendor ARiC16

Reducing system complexity: A consequence of Selecting fewer components ARiC15 can be seen to contribute to reducing System complexity because this can contribute to reducing the number of separate connections between components. Choosing components supplied by same vendor ARiC16 implies selecting components which are built to integrate with each other because they support the same architectural standard. Again, selecting components which are more likely to integrate with minimal effort because they support the same architectural standard can be seen to contribute to reducing System complexity because there is less likelihood of integration problems occurring.

Comparing Sourcing components from same vendor ARiD1 with Acquiring pre-integrated components ARiD9

It can be seen that Sourcing components from same vendor ARiD1 and Acquiring pre-integrated components ARiD9 can contribute to reducing System complexity because with pre-integrated components there are fewer interfaces needing to be managed by system developers (the assumption is that the number of interconnected interfaces contributes to system complexity because each separate interface requires effort by system developers to connect, integrate and manage. With pre-integrated components the connection, integration and management effort of the set of components comes under the vendor’s responsibility – system developers only have to manage their integration with the rest of the system).

Comparing Saving development time PMiA2 and Preferring large components PMiA4

From the data code, Preferring large components PMiA4, expresses COTS system developers’ preference to use fewer larger components, which satisfy some of the system requirements, rather than selecting greater numbers of smaller components which can deliver greater proportions of system requirements. The assumed consequence arising from Preferring large components PMiA4 is to contribute to reducing System complexity as a result of a lesser number of interconnected interfaces.

Therefore, the relationship between codes Saving development time PMiA2 and Preferring large components PMiA4 can be seen to be a Cost reducing strategy arising by choosing to build systems from COTS components in order to reduce development time and from system developers reducing System complexity by selecting fewer, larger components to reduce the number of interconnected interfaces.

Comparing Maintaining multiple interfaces PMiA3 and Increasing system
A consequence of maintaining multiple interfaces can be increasing system maintenance effort for system administrators and maintainers as a result of increasing system complexity as the number of interconnected interfaces between components also increases.

A result of integrating many components in a COTS-based system can be a greater number of interface connections between components compared with systems constructed from fewer components.

Comparing code: System complexity ARiL21

The suggestion from the data is that the system complexity of COTS-based systems can affect aspects of their cost. System complexity was defined as the number of parts, the number of connections and the number of dependencies between system parts. Thus, systems performing complex tasks, comprising of many different components, each with numerous configuration settings and various instances of integration code and higher degrees of dependencies can be associated with higher degrees of system complexity, compared with systems with fewer instances of these variables. A consequence of increasing numbers of variables requiring consideration during system development can be increasing effort and skill levels, greater likelihood of problems occurring following maintenance tasks and more effort to resolve any issues.
Memo on Concept Vendor homogeneity - PMiA2, PMiA9, ARiB14, ARiB15, PMiG12

Introduction

**Vendor homogeneity** implies a commonality of components produced by the same vendor. The assumption by system developers is that components produced by one vendor will support the same architectural standard and be designed to integrate and work together with minimal effort.

A further assumption is that vendors will manage and test the ongoing maintenance of their components to ensure that they continue to function together.

It can be seen that **Vendor homogeneity** is a design decision when components are acquired from the same vendor because they are considered to integrate and function together.

However, the data indicates that there are *conditions* associated with **Vendor homogeneity**. As vendor organisations become larger and release more products the complexity of their product support structure can increase. In some organisations products may be developed and supported by different teams. Furthermore, in some organisations the different teams do not necessarily cooperate with each other. Thus, the experience of customers attempting to receive support for multiple components supplied by the same vendor can be poor when the vendor’s support organisation lacks cooperation and synergy.

Therefore, a consequence of **Vendor homogeneity** can be reducing costs because the effort required for integrating and maintaining components supplied by the same vendor should be less then the effort required to integrate and maintain components supplied by different vendors.

**Vendor homogeneity** is a design decision applying the DESIGN PRINCIPLE of selecting components which are likely to integrate easily together.

**Vendor homogeneity** emerged from the following code comparisons:

Comparing **Saving development time PMiA2** and **COTS supplier issues PMiA9 Revised**

From the data, code **COTS supplier issues PMiA9** relates to the assumption that COTS components supplied by the same supplier, **vendor homogeneity**, will support the same architectural standard and components supplied by different vendors may not (this may not always be the case).

Therefore, a consequence of choosing components supplied by the same vendors, supporting compatible architectural standards, is **Saving development time PMiA2** because it is assumed that these components are designed to integrate with minimal effort.

Furthermore, the assumption is that less integration effort is required for components supplied by the same vendor because they support the same architectural standard and are designed to work together.
Comparing **Homogeneity of vendors ARiB14** with **Vendor’s organisational complexity ARiB15**

In interview A (PM) *Homogeneity of vendors ARiB14* was seen as being beneficial to COTS-based design because the assumption by system developers was that components supplied by the same vendor would support the same architectural standard, requiring less integration effort, thus, contributing to reducing costs.

However, from Interview B (Architect) *Homogeneity of vendors ARiB14* does not always contribute to reducing costs. For example, when vendors organisational structure are complex there can appear little synergy between internal departments for customers requiring product support. In some cases, different departments can appear as separate companies to the customer. The result is that service can be as bad as when dealing with different companies. Therefore, the consequence can be poor support and cooperation for customers who have acquired components supplied by the same vendor.

Therefore, the assumption that **Vendor homogeneity** is always a beneficial **DESIGN PRINCIPLE** can be challenged when vendors’ organisations are too complex to provide good customer support.

Therefore, in this case system developers may be left with **Balancing [the] design principles** of the benefits of vendor homogeneity with deficiencies of complex vendor support organisations.