An Assessment of Existing Component-Based Software Development Methodologies and a Holistic Approach to CBSD

Thein Than Tun

A thesis submitted in partial fulfilment of the requirements of
London Metropolitan University
for the degree of
Doctor of Philosophy

May 2005

ANY MAPS, PAGES, TABLES, FIGURES, GRAPHS OR PHOTOGRAPHS, MISSING FROM THIS DIGITAL COPY, HAVE BEEN EXCLUDED AT THE REQUEST OF THE UNIVERSITY

Abstract

Software applications nowadays are getting ever larger and more complex. At the same time, users and sponsors of software applications have increasingly higher expectations: lower development cost, faster delivery time and higher quality of the products. This creates new challenges that the traditions of developing a software application from scratch every time a need arises, and of reusing code at a low-level of programming, are unable to address adequately. Component-Based Software Development (CBSD) is a strategic attempt to address these challenges by promoting extensive software reuse throughout the software development stages. This development strategy raises a spectrum of important issues. This research is primarily concerned with methodological issues such as system modelling, architecture and development process. This research validates two main hypotheses. The first hypothesis is that the theoretical basis of existing CBSD methods is weak. To test this hypothesis, existing CBSD methods need to be evaluated using an extensive and rigorous evaluation approach. This research identifies four publicly available CBSD methods and numerous approaches to evaluation of system development methods. These evaluation approaches are deemed unsuitable for the kind of evaluation envisaged by this research. Therefore, a new comprehensive framework for evaluating system development methods, called the MAP framework, is proposed. The existing CBSD methods are then evaluated using the MAP framework, which confirms the first hypothesis. The second hypothesis is that the limitations of the existing CBSD methods can be overcome. To test this hypothesis, various positive features of existing CBSD and non-CBSD methods that the MAP framework helps identify are synthesised, giving rise to NAVITA, a holistic CBSD method proposed by this research. The new CBSD method is then evaluated using the same criteria and rigour applied to existing methods. The evaluation confirms the second hypothesis. Furthermore, this research contributes to the application of Object-Oriented system development methods by proposing a set of principles that govern a rational allocation of class operations.

Acknowledgement

I would like to express my sincere gratitude to my supervisors, Dr. Peter Bielkowicz and Preeti Patel – both of the Department of Computing, Communication Technology and Mathematics, London Metropolitan University (London Met) – for their generous help, wise guidance and invaluable support. It has been a great pleasure to work with them both. Special thanks to Dr. Peter Bielkowicz for giving me this wonderful research opportunity after my first degree.

Many thanks to Dr. Islam Choudhury (London Met), Dr. David Soud (Kent University) and Vic Page (London Met) for sharing ideas and to all QC309 students (2003-2005) for participating in IPI Matrix experiments.

Thanks also to my teachers in Burma, who include the late U Aung Kyaw Myint, U Sann Lwin, Daw Thida Aung of MCC and U Thaung Tin of KMD.

Special thanks to Zibby for proofreading the whole thesis. Thanks also to my flatmates Ma Mawgyi, Mg Mawgyi, Phosa and Sumi for all their kindness in creating a comfortable environment in which to write up this work.

My family have given me the best possible moral and financial support for my studies in the UK. Without their encouragement, I would not have been able to complete this work.

Table of Contents

List of Figures

List of Tables

List of Charts

Chapter One

Research Overview

1.1 Introduction

Creation of the Unified Modelling Language (UML) has marked the end of a tremendous transformation in Software Engineering (Booch et al, 1999; Rumbaugh et al, 1999; UML Success Stories, n. d.). It effectively ended the era of what is wittily called the 'method war', in which method experts promoted different brands of OO methods over Structured and other OO methods (Fowler and Scott, 1997). UML seems to have had the last word on OO methods. It is, perhaps, an indication that OO methods have come of age.

It has long been maintained that one of the main strengths of the OO approach to system development over its predecessors is better software reuse (Jacobson et al, 1992; Coad and Nicola, 1993). Class inheritance or generalisation/specialisation mechanism allows developers to create new classes by reusing the existing ones. However, there was a recognition that in order to yield substantial reuse, developers need to look beyond the reuse of individual classes (Udell, 1994; Aoyama, 1998a and 1998b; Brown, 1996). As the debate over OO methods has settled with the arrival of UML, another approach to software development with an even greater emphasis on reuse has emerged: Component-Based Software Development (CBSD). Accumulated interests, both academic and commercial, in CBSD is evident from the growing list of published works on this topic as well as increasing commercial availability of components and component technologies. For example, the ComponentSource website, a popular online marketplace for component buyers and sellers, claims to be in partnership with over 500 component publishers worldwide (ComponentSource, n. d.). Most standard texts on Software Engineering include chapters on the topic (Pressman, 2005; Sommerville, 2004).

generated excitement amongst software developers (Kozaczynski and Booch, 1998). Bashir (2003) for instance, argues that even though CBSD is still evolving, there are commercial imperatives for adopting the development strategy now, while predicting greater availability of commercial components in future.

1.2 Component-Based Software Development

CBSD, often loosely referred to as Component-Based Development (CBD), is a relatively new software development strategy. Traditional development approaches are criticised for implicitly encouraging the reinvention of old wheels and not giving enough emphasis to the substantial reuse of software artefacts in all development stages. Its main proposition, originating from electronics and computer hardware engineering, is that applications need not be monolithic and completely unique. Instead, they may be assemblages of loosely-coupled components, where interfaces are clearly defined and implementations are well-hidden (Szyperski, 1997). With CBSD, every time a need for a new system arises, the first response is to investigate whether an existing application can be deployed as it is or adapted, without needing to carry out all the traditional software development activities. Developers may look for components in a range of sources including Commercial-Off-The-Shelf (COTS) packages, legacy systems, and nowadays, online component vendors (Allen and Frost, 1998). With this approach, creation of new applications is considered only if the reuse of the application through adaptation and negotiation and adjustment of requirements is not feasible. This is therefore, a "reuse first" approach (Ambler 1998 and 1999). Catalysis (D'Souza and Wills, 1999) defines CBSD as follow.

> An approach to software development in which all artefacts – from executable code to interface specifications, architectures, and business models and scaling from complete applications and systems down to small parts – can be built by assembling, adapting, and "wiring" together existing components into a variety of configurations.

CBSD is often portrayed as a natural successor to OO development. Indeed, most CBSD methods, techniques and technologies have evolved from OO counterparts. A fusion of OO concepts with contemporary developments in system development methods, such as domain modelling and design patterns, and software technologies such as middleware architecture, plug-in technologies and component packaging

mechanisms in various programming languages have thrust CBSD into the forefront of modern software development approaches.

1.3 Research Objectives

Since the late 1990s, some systems development methods purported to be based on the CBSD approach have appeared in publications. An initial investigation by this research suggests that these methods have major weaknesses in many aspects and that these weaknesses can be overcome. The main aim of this research is to contribute to the advancement of CBSD methods by means of these two objectives:

(a) following an analytical survey, a critical evaluation of existing CBSD methods using a rigorous evaluation framework

(b) creation of a new holistic approach to CBSD that overcomes weaknesses in existing methods

1.4 Research Context

CBSD has a number of important and related dimensions including:

- Business Dimension – mainly concerned with the implications CBSD has on the business processes, development time and cost, and general effectiveness of the approach (Jacobson, 1997; Veryard, 1998)
- Technological Dimension – concerned with issues such as the availability of the necessary component-based technologies, their compatibility, reliability, reusability etc (Szyperski, 1997)
- Economics Dimension – involves financial planning and investment strategy for large scale reuse of components in long term (Aoyama, 1997; Sametinger, 1997)
- Legal Dimension – legal considerations for ownership, rights to modification, and liability of reusing components developed and/or integrated by other parties (Chávez et al, 1998; Yoche, 1989; Yoche and Levine, 1989; Jakes and Yoche, 1989)
- Methodological Dimension – technical aspects of analysing, designing and implementation of component-based applications (Jacobson, 1997; D'Souza and Wills, 1999)
- Project Management Dimension – planning and control of component-based projects (Jacobson, 1997; Atkinson et al, 2002)

The methodological dimension of CBSD is the principle interest of this research, in particular, System Modelling, Software Architecture and System Development Process. The two stages of this research are outlined below.

1.5 Stage 1 – Evaluation of existing CBSD methods

This stage of the research is mainly related to a rigorous theoretical evaluation of existing CBSD methods. It is important here to emphasise the nature of evaluation this research is concerned with because it has significant relevance to the research methodology used.

Avison and Fitzgerald (1995 and 2003) suggest that there are two strongly connected reasons for evaluating methods: academic reason and practical reason, where the former attempts to understand the philosophy and nature of method in order to classify and improve them, whilst the latter deals with the question of applicability of some method(s) within a limited context.

The nature of evaluation in this research is academic in the sense that it aims to critically examine the technical quality, and consistency of CBSD methods. It means for example, examining the systemic coverage of various models provided by a method, the rigour of modelling techniques, and consistency between models. The main purpose therefore is not to verify claims of methods, as in "Does the method do what it says it will do in practice?", but to examine the basis on which claims are or can be made, as in "Does a method have good enough reasons to claim what it claims, generally that it is good?" Therefore, the research aims to examine the strengths and weaknesses of the principles, or lack of the principles, on which the existing methods are built and how these principles are manifest in the methods.

It is important here to note the difference between empirical and metaphysical approaches to explaining physical phenomena. Lowe (2002) explains that empirical sciences such as physics are concerned with 'explaining certain basic and ubiquitous phenomena in the natural world, that is, in the realm of things existing in space and time.' Metaphysics is not only concerned with the nature of physical entities but also 'with the nature of space and time themselves, and with the nature of causation.' In addition, metaphysics is also concerned with abstract entities that do not exist in space and time such as numbers and sets, and entities that exist in space

and time, such as mental states of thought and feelings of people, the behaviour of which according to many philosophers and scientists, 'can never be explained solely by appeal to the laws of physics, not least because their behaviour is, in large measure, subject to *rational* rather than merely to *causal* explanation'.

The objects of study in this research such as modelling concepts, techniques, consistency between models and so on, are by nature abstract entities, like our thoughts and emotions. Therefore, any serious evaluation of their inherent qualities can only be an essentially rational exercise.

Such an enquiry helps both creators and intelligent practitioners of methods determine the quality of a given method. Moreover, it provides the necessary further knowledge for improvement of methods. Such knowledge of methods, crucial for the next stage of this research, cannot be ascertained from mere statistical analyses of empirical data. Therefore, this evaluation will not be overtly concerned with matter-of-fact issues such as how improvement of software quality is facilitated by a given method. Rather, the research is focused on a rational inquiry into the quality of CBSD methods.

In order to evaluate existing CBSD methods, this research investigates evaluation approaches used by researchers and practitioners. A survey of these approaches, described in Chapter 2, shows that there is no effective means for understanding and evaluating system development methods. Approaches to evaluating CBSD methods in a manner envisaged by this research are non-existent. In this research, a novel approach to evaluating system development methods, partly based on the well-established theory of systems thinking and systemic evaluation approaches, is advanced (Chapter 3). Using this new evaluation framework, existing CBSD methods, Software Reuse (Jacobson et al, 1997), Select Perspective (Allen and Frost, 1998), Catalysis (D'Souza and Wills, 1999), KobrA (Atkinson et al, 2002), are evaluated. Another method, SCIPIO, initially considered for evaluation, has been dropped because development of the method by its authors was abandoned. A summary of the evaluation and results of a small experiment about the repeatability of the evaluation framework are presented in Chapter 4.

1.6 Stage 2 – Creation of a new CBSD method and its evaluation

A systematic evaluation of existing CBSD methods in the previous stage exposed their relative strengths and weaknesses. Furthermore, the evaluation framework used also indicated how either existing methods could be improved or a new and better method created. This critical knowledge of the existing methods led to the creation a new CBSD method by synthesising strong features of the existing methods and novel insights into CBSD. Again, the new method deals with the three main, and related, aspects of CBSD methods, namely, System Modelling, Software Architecture and System Development Process (Chapter 5 to Chapter 13). The new method is then evaluated using the same framework used to evaluate existing CBSD methods before comparing the evaluation results (Chapter 14).

Chapter 15 describes the methodology of this research, whilst the research conclusions, contributions and areas for further research are discussed in Chapter 16. There are seven appendices in this thesis:

- Appendix I lists a full glossary of acronyms and abbreviations used in this thesis
- Appendix II presents a journal-formatted paper surveying existing CBSD methods
- Appendix III presents another journal-formatted paper discussing the principles for rational allocation of class operations
- Appendix IV illustrates the proposed CBSD method using the LibrInfoSys case-study
- Appendix V shows a comparison of a structured diagram with an activity diagram
- Appendix VI contains the raw data gathered from the experiment described in Section 4.6

1.7 Language, Terminology and Abbreviations

British English spellings are used throughout this thesis. For consistency, original quotations are transcribed into British English (CIDE, 1995).

When referring to system development approaches, the term 'method' is used over 'methodology'. Although methodology was the original choice of word in

documents such as the project proposal form, and the two terms are often used interchangeably by other authors (Jayaratna, 1994), in this thesis, 'methodology' is used only to mean 'research methodology'.

The terms 'system', 'software' and 'application' are used in the following broad sense: system is mainly used at the analysis stage when the exact solution to the users' problems is not known; software is the generic term for the solution in the design sense; and application is the implementation of the solution. A full list of commonly used acronyms and abbreviations is provided in Appendix I.

Chapter Two

Approaches to Evaluation of SDMs

2.1 Introduction

It is a widely accepted view in science that the method of inquiry is more important than the inquiry outcome. Results alone are meaningless unless the method used is sound (Brown et al, 1989). Similarly, when evaluating system development methods or creating a new method, it is important to scrutinise the evaluation or creation process before accepting the conclusions drawn.

A literature survey carried out in this research reveals that there are a number of approaches that may be used to evaluate products, tools, technologies and methods. These approaches range from feature analysis to benchmarking, from subjective to statistical analysis, and from empirical to downright arbitrary approaches, including many imaginable shades of grey between them[1] (Galliers and Land, 1987; Basili, 1993; Kitchenham and Pickard, 1995; Evaristo and Karahanna, 1997; Zelkowitz and Wallace, 1998; Basili et al, 1999). The evaluation approaches are diverse not only because there are relative strengths and weaknesses in each of these approaches, but also because the purpose of evaluation is often very different. For example, commercial software developers may be more interested in the market shares of the methods (Hutt, 1994), while academic researchers will be more concerned with the

[1] One remarkable finding of this survey is that most discussions on research methodologies are related to validation of tools, technologies and methods. No research methodology for creation of SDMs has been found. Even authors of new SDMs do not generally discuss how their methods have

technical qualities of the methods (Bubenko, 1986). Such differences in nature of interests in methods and purpose of their evaluation affect the choice of evaluation approach. If the purpose of evaluation is to establish the market share of two competing methods, methods such as survey may be appropriate because this requires eliciting responses from a number of people spread over a large geographical location. On the other hand, if the purpose is to establish how a given method can help improve certain quality attributes of the end product, statistical analysis of empirical data will be more appropriate.

Avison and Fitzgerald (1995 and 2003) catalogue a long collection of approaches proposed and used over many decades for evaluation of SDMs. There are notable omissions in the list, but it is beyond the scope of this project to engage in detailed discussions about the issue.

An approach for evaluation of SDMs has to provide two key elements: criteria and evaluation process. The "criteria" define the important or desirable quality attributes of SDMs and the "evaluation process" suggests how these quality attributes of SDMs should be observed and measured. If the evaluation is concerned with a particular product such as a CASE Tool, or with quality characteristics that are obvious and directly measurable such as "efficiency" of a particular design algorithm, then formulation of quality criteria and the evaluation process will be relatively straightforward. SDMs neither have dominant quality attributes that determine the overall quality of the methods, nor are their quality attributes always directly measurable. Therefore, a rigorous SDM evaluation approach must provide a mechanism for formulating a set of evaluation criteria that collectively determines the general quality of a method, and an outline of the process through which the quality of an SDM can be determined.

The following sections provide an overview of a wide variety of evaluation approaches in order to discuss their relative merits, their potential effectiveness and applicability of these approaches to this research. The discussed evaluation approaches are carefully selected – not to be comprehensive, but to be representative

---

arisen. For that reason, discussions will largely focus on approaches to evaluation, rather than creation, of methods.

of the diversity of ideas – so that different important ideas relevant to this research can be illuminated. The evaluation approaches are divided into two main groups: empirical approaches and non-empirical approaches (Van Horn, 1973).

2.2 Empirical Approaches

It is tempting for researchers to try and support their speculations – alas, all *good* hypotheses and theories are speculative by nature; indeed if they do not speculate, they would not tell anything that is not already known[2] – with statistical analyses of empirical data of varying forms and sizes. Basili et al (1999) give the following definition.

> An empirical study, in a broad sense, is an act or operation for the purpose of discovering something unknown or of testing a hypothesis, involving an investigator gathering data and performing analysis to determine what the data mean.

Empirical approaches cover various methods, such as case study, field experiment, field study (survey), and laboratory study, qualitative study and archival analyses (Basili et al, 1999; Van Horn 1973).

However, researchers in various fields of computing have a reputation for *not* using empirical research approaches. For example, a survey 600 research papers published in 1985, 1990 and 1995 show the use of research methods that do not have empirical rigour are prevalent (Zelkowitz and Wallace, 1998). Some consider this an extremely serious problem. Tichy (1998), for example, passionately argues that "computer scientists" should experiment more, and he attempts to refute various "fallacies" about why empirical methodologies are not widely used. Kitchenham is a well-known advocate of statistical analysis methods in software engineering with her work such as (Kitchenham, 1992; Kitchenham et al, 1995; Kitchenham et al

[2] Popper (1972) has suggested that scientific theories are distinguished from others by their refutability. That is, a given theory can be accepted not only because there is evidence to support it, but perhaps more importantly because there is no available evidence against it, and that it is clearly possible for the evidence to become available and if that evidence were to become available, it will falsify the theory. Therefore, every scientific theory must contain an element of 'risk' that it will be falsified; because a theory must predict what is not known, it is speculative.

2002). Pfleeger also strongly promotes the use of empirical approaches in her work such as (Pfleeger, 1995).

There are those who strongly doubt the necessity of empirical approaches to computer science. Tichy himself cites Fred Brooks as saying that computer science is "not a science, but a synthetic, an engineering discipline" and the current editor-in-chief of IEEE Software, Warren Harrison, as suggesting that "gut feeling is enough when adopting new software technology; experimentation and data are superfluous." Even where there is an agreement on the need for empirical methodologies, there is no consensus on what they should constitute.

This research is not concerned with whether or not scientific methodologies are applicable to research in computer science, this is too wide a field for comment. Rather it is concerned with their applicability to research in SDMs. White (1982), as cited by Avison and Fitzgerald (1995), argues for the use of scientific methods in evaluation of SDM because only repeatability of a method justifies its adoption. A notable rejection comes from Checkland (1987), who sets two challenges: in cases of successful application of a method, he demands proof that another method could not be successful; in cases of failures, he asks for proof that it is not the incompetence of developers that is really responsible. Neither case has been proven, according to Avison and Fitzgerald (1995).

2.2.1 Formulation of Criteria

Empirical approaches are exclusively concerned with the process of observation, testing and validation, and not what should be observed, tested and validated. Therefore, evaluation criteria need to be formulated before any empirical approach can be applied.

2.2.2 Evaluation Process

If a genuine empirical approach is to be used, the following will be an appropriate research methodology:

- Apply each of the existing CBSD methods to a real-life project for the same or near-identical system, involving the same developers or developers with near identical profiles.

- Measure whether the system was developed on time, within budget, to the quality acceptable by its users, and other stakeholders who have vested interests in the system: precise criteria need to be known.
- Repeat this exercise in projects of different types and sizes, and in different implementation environments. As the number and diversity of projects in which the methods are applied increase, more confidence can be attached to the conclusions drawn from analyses of the accumulated data as to which method is better.
- If the evaluation of the measurements shows that the existing methods are unsatisfactory, synthesise various good elements from the existing methods to create a new one and reapply the new method in the same context. Again, measure how the new method performs and compare the measurements to those made for the existing methods. From these results, draw conclusions about the quality of the new method against the existing ones.

It is clear from this account that if such an investigation is undertaken, it will contribute hugely to our knowledge about the effectiveness of the use of various system development methods in general, of which little is known at present[3] (Avison and Fitzgerald, 1995), and of individual methods, of which even less is known. As noted by Tichy, the same can be said about programming paradigms, and many other aspects of computer science. Such research will have to be done on a very large scale, over a long period of time, involving numerous people with diverse professional expertise, costing a large sum of money and indeed, would be truly revolutionary.

If this is the general standard for a method to be acceptable, almost all published methods fall short since not a single method in the public domain has such empirical meticulousness. This does not mean that a given method has not been applied in a great number of projects. The point is that system development methods

[3] At an international workshop the author attended, the same question was asked, and was met by a certain amount of despondency. Nobody had any answer. An IBM whitepaper (Cernosek and Naiburg, 2004), for example, claims that modelling helps reduce technical and financial risks, but it does not offer hard evidence. Judging by the popularity of UML, there might be some truth in this claim.

traditionally do not have an empirical justification. In fact, they tend not to be supported by any justification at all. For example, most component-based methods investigated in this research justify themselves by neither serious arguments nor empirical evidence. The same is true for most OO methods and even UML. This does not, of course, mean there should be no such research to find out if they actually work; the reverse is the case.

There are some practical difficulties with the controlled experiments required by these approaches. It is reasonable to expect a developer to know one or two popular methods, but there are unlikely to be many developers with detailed knowledge of all of the existing CBSD methods. To overcome this difficulty, groups of developers with different expertise may be used; having to take into account the differences in developers' profiles causes an extra problem. From the time the first method is applied, developers' familiarity with, and knowledge of, the system will increase very rapidly, which will favour the application of later methods. Against this factor, developers may also become weary of working on the same system repeatedly, which can inevitably adversely affect their quality of work. Jayaratna (1994) for example identified a number of personal factors, such as experience and prejudice, which affect how a method is used to solve a problem. In a sense, there is a complex web of psychological influences at work, which add additional difficulties to the measurement of how well the method has performed (Basili et al, 1999).

Suppose a group of super-developers who can work as necessary can be found and they develop the application using each of the methods. The question now is: what should be measured? What are the quality attributes that will help decide the quality of the methods used? Sponsors of the projects will be interested in delivery time, budget, and general user acceptance of the application. What about the technical qualities of the application itself? Whatever the outcome of the project, it will not be clear if it was the development method used, and not say the programming experience of developers, that has mainly contributed to the success or failure of the project. It could be both. How can one know how much each has contributed? If the quality of the application itself is going to be measured, what exactly should be measured? What if the nature of applications is hugely different? Some of these

quality criteria will clash; shorter development time and lower cost may negatively affect the technical qualities of the product.

Suppose it is still possible to do this exercise in a project. The process needs to be repeated across different projects involving different set of people, for different types of application, in different environments and so on, before any conclusive verdict can be reached.

Limitations of the empirical approach in relation to the creation of the new method are even more acute. Inventing new methods, by definition, is a creative process that requires a leap of thought. Philosophers of science, such as Popper, suggest that empirical evidence does not lead one in a linear fashion towards a valid generalisation. Pfleeger (1999) makes clear that applicability of empirical approaches to software engineering is limited.

Such a type of empirical research, in terms of resources, is beyond the reach of individual academic research projects. In these projects, it is a reasonable aim to evaluate CBSD methods in order to understand what makes a method good through rational means (Section 1.5), rather than being concerned with the mammoth task of discovering whether the methods measure up to their claims in practice. For this reason, the use of empirical research approaches to evaluate existing methods in this research is not called for.

2.3 Non-Empirical Approaches

Most evaluations of system development methods do not use empirical approaches, and if they do, the criteria are usually narrow. For example, most evaluation approaches surveyed by Avison and Fitzgerald (1995) are non-empirical in nature.

2.3.1 Subjective Criteria Approaches

There are too many suggested criteria to be listed here comprehensively. Most of these criteria are characterised by limited scope, subjective application and often random organisation. Authors of some approaches do not explain why the criteria are important, or even what they ultimately aim to achieve. The use of such criteria is acceptable where the focus of evaluation is limited.

For example, authors of a new ambitious object-oriented method called ADORA (Glinz et al, 2001) attempt to show that their method is superior to UML by analysing fifteen people's responses to questions regarding "two fundamental qualities" of a specification language: comprehensibility ("a specification must be easy to understand") and acceptance ("user must like it").

However, there are more sophisticated approaches. Avison and Fitzgerald (1995), for example, provide a set of seven main criteria, which include Philosophy, "a principle, or a set of principles that underlie the methodology"; Model; Techniques and Tools; Scope; Outputs; Practice; Produce. Some of these criteria are further broken down; for instance, Philosophy involves Paradigm, Objectives, Domain and Target.

As cited by Avison and Fitzgerald (1995), Catchpole (1987) summarises "the views of a number of authors concerning the important areas of concern when comparing methodologies" and suggests a set of twenty three criteria that include Rules, "formal guidelines in a methodology to cover phases, tasks and deliverables, and their ordering, techniques and tools, documentation and development aids, and guidelines for estimating time and resource requirements"; Total Coverage; Teachability. Land (1982) adds three more criteria to the list: "A systematic way of looking into the future"; "The integration of the technical and non-technical systems" and "Scan for opportunity". Not to be outdone, it seems, Avison and Fitzgerald (1995) further append the following criterion to the list: "Separation of analysis and design".

Bjorn-Andersen (1984) has created a checklist including questions such as: "What is the context where a methodology is useful?", "To what extent is modification enhanced or even possible?" and "Is user participation *really* encouraged or supported?"

2.3.1.1 Formulation of Criteria

There are a number of problems with these approaches. Some criteria are too generic, bordering on vacuity. What is needed to be compared in models; symbols, concepts or something else? Organisation of the criteria is often arbitrary. For example, where expert opinions are drawn upon, how are "experts" selected? What

"experts" they change views or disagree with each other? If experts can add whatever criteria they think are important, when will the list end? And will an endless list of criteria help evaluators determine the quality of a method? This does not mean that some of the suggested criteria cannot be used; rather, the way in which the criteria are articulated, organised and applied does not instil objectivity, comprehensiveness and authority. Even a seemingly self-evident and commonly suggested criterion such as simplicity is problematic. The level of simplicity of a method does not necessarily bear any relation to the overall quality of a method. In any case, how should one measure the simplicity of a method? If the simplicity is to be considered collectively with other criteria, there remains the question of how this should be done.

2.3.1.2 Evaluation Process

These criteria only suggest what to evaluate in general terms, not how to carry out the evaluation. How can different philosophies of methods be evaluated? How are models assessed? What if they are based on different paradigms, for example, in the case of Data Flow Model, which is based on the structured paradigm and Use Case Model, which is based on the object-oriented paradigm?

It is clear that the use of a random list of criteria is unlikely to lead to the interesting discovery of qualities of system development methods, unless the remit of the research is narrowly defined. Therefore, the use of such a set of criteria is not justifiable for this research.

2.3.2 Meta-Modelling Approaches

There is an increasing tendency for authors of new methods to use meta-models as, or as part of, their justifications of their proposals. For example, methods/modelling languages such as Open (Firesmith et al, 1997; Graham et al, 1997; Henderson-Sellers et al, 1998;), UML (OMG, 2003) and KobrA (Atkinson et al, 2002) provide various meta-models. Hong et al (1993) describe a two-phased evaluation approach, based on meta-modelling, to compare OO methods.

In Phase One, two meta-models are built: one is the **meta-process model** of analysis and design steps, including input and output from each step, and the other is the **meta-data model** of concepts and techniques showing "both the definitions of the

concepts and the relationships among them." Hong et al do not discuss how the meta-models have been built; they only give examples of a meta-data model and a meta-process model of an OO method described by Wirfs-Brock et al (1990).

In Phase Two, the methods are compared. The comparison of processes begins by creating a "supermethodology", which is "the smallest common denominator of all activities depicted in the meta-process models" of all investigated methods. A table is produced listing all activities of methods against the supermethodology and showing how they correspond. Concepts from the meta-data model are compared following the same approach. A set of concepts of supermethodology from the meta-data models is drawn, against which concepts from methods are compared. In addition to the "string" indicators used in the previous table, "numbers" are also used to provide a footnote to the concept.

2.3.2.1 Formulation of Criteria

The supermethodology, created from concepts and development activities of existing methods, serves as the evaluation criteria. The main difficulty with the philosophy of this approach is the existing methods are used as a basis for deriving criteria used to evaluate the same methods; therefore, if all the existing methods are flawed, any common denominator produced will also be flawed. Therefore, the supermethodology does not lend itself as a standard against which others should be measured.

2.3.2.2 Evaluation Process

Hong et al (1993) make some attempt to quantify the quality attributes. There are also problems here. For instance, when comparing techniques, the approach does not suggest how techniques used to capture objects should be compared; rather it superficially suggests that evaluators should mention the name of techniques.

2.3.3 Action Research

It can reasonably be argued that the description of the empirical approach in Section 2.2 might be rather extreme; perhaps a watered-down version of the approach, a kind of empirical-*light*, could be applied instead. One such approach popular with IS researchers is the action research approach (Wilson, 1984; Galliers and Land, 1987;

Avison et al, 1999). This approach emphasises the participation of the observer in the phenomenon or the process being studied, so that he or she is no longer a fly-on-the-wall witness but an active learner interacting with the subject(s), guiding or even working with the subject to arrive at the desired solution. Based on such experiences, the observer theorises and tests dynamically his or her theories. For example, the author of NIMSAD, Jayaratna (1994), claims that his framework is based on his personal experience of working with clients as a consultant and his own action research. There are those, such as Avison and Fitzgerald (1995), who believe that this may be the only appropriate approach to IS research.

2.3.3.1 Formulation of Criteria

Since the emphasis of the action research approach is the exploratory and cyclic process of learning, reflection and generalisation, the formulation of criteria is subjective, dynamic and highly subjective.

2.3.3.2 Evaluation Process

The evaluation process involves the research working closely with those who are involved in the "problem situation," and developing an interpretivist narrative to explain a phenomenon.

This approach may be appropriate if the project is concerned "soft" issues such as human factors in IS projects. The main problems with this approach include its subjectivity, repeatability, and time.

Simply because a method has been evaluated through action research need not mean that the results will be objective. The exercise needs to be repeated across a number of projects before any reliable conclusions can be drawn. Since only one method can be evaluated in a project, evaluation of multiple methods will require a number of projects to be carried out. Still, the verification would not be independent.

Other problems with the methodology are concerned with the method user(s), the person(s) who applies/apply the new method. There are questions about who should apply this method in a real-life project. The author, with intimate knowledge of the method, is arguably in the best position to be the method user. However, this is unrealistic because medium-sized software development projects need more than

one analyst. Furthermore, this gives the author's method an unfair advantage over other methods. It is however possible for other developers in the project to be trained in this new method so that they can also apply it. There are issues about who should apply it and when they would be ready to apply.

For these reasons, the action research approach is not appropriate for this project.

2.3.4 Systemic Approaches

There are also evaluation approaches that are based on the concept of systems, and systems thinking, that generate questions regarding various qualities of methods. Two systemic approaches, proposed by Wieringa (1998) and Jayaratna (1994), are discussed in this section.

Wieringa (1998) discusses a comparison framework that is based upon the concept of systems that interact with their environments. These interactions can be grouped into meaningful units called *functions*. Functions have two orthogonal properties known as communication (interaction) and behaviour (time-dependent communication). Wieringa regards functions, communication and behaviour as "system properties" that can be described at various levels of abstraction. System interaction at the top level can be called *mission* of the system, at a more detailed level, functions, and at the bottom level, *atomic transactions*. Also, behaviour of the system can be described at various levels. A refinement hierarchy is used to show the relationship between various levels of description.

Figure 2.1 The magic square (Wieringa 1998)

Systems are composed of parts, and their composition is shown in aggregation hierarchy of systems. Hierarchies of aggregation and refinement are also intrinsic properties, and can be represented in a magic square. Wieringa writes:

> At a given level of aggregation and abstraction, we can decrease the abstraction level at which we specify the interaction of a system at that level without decreasing the aggregation level. ... Conversely, at a given level of aggregation and abstraction, we can decompose a system without decreasing the level of abstraction. ... We allocate a system interaction to one or more components if we decide that these components will realise the system interaction.

System decomposition and interaction refinement lead to the idea of Function Decomposition Table, which shows the mappings of system interactions onto system components.

Figure 2.2 Function Decomposition Table. (Wieringa 1998)

Based upon the concepts discussed above, Wieringa concludes that a method should offer (modelling) techniques for four properties:

- function specification techniques
- behaviour specification techniques
- communication specification techniques
- decomposition specification techniques

Since Wieringa regards decomposed system parts as systems in their own right, he has come up with the following criteria. A method should have specification, which in most cases implies modelling, techniques for External Communication; External Behaviour; External Function; Conceptual Decomposition; Component Functions; Component Behaviour and Component Communication. These criteria provide the systematic and systemic basis for determining the sort of models the methods surveyed in his paper should offer.

Jayaratna (1994) proposes the Normative Information Model-based Systems Analysis and Design (NIMSAD) framework, which is a generic framework for the

evaluation of *any* method, including Information Systems Development methods. He suggests that effective application of a method depends upon three elements: the method itself, the person who applies the method, and the context in which the method is applied.

Problem Context

NIMSAD suggests that if a method is to be evaluated, there is also a need to evaluate the organisational context in which the problem exists, and to which the method is applied. The primary reason for this is that the ultimate test for the method is to demonstrate its effectiveness when applied to a problem situation, which can only be observed in the changes it brings to that situation.

Intended Problem-Solver

Effective application of a method is subject to the "personal characteristics" of those who apply it in their given situation. These include: Perceptual Process; Values; Ethics; Motives; Prejudices; Experiences; Reasoning Ability; Knowledge and Skills; Structuring Processes; Roles; Models and Frameworks.

Problem Solving Process

The method itself is regarded as the problem-solving process. This is broken down into three phases, which are further divided into stages.

- Phase 1: Problem formulation
  - Stage 1: Understanding the situation of concern – Before diagnosing the problem, there is a need to grasp the problem situation.
  - Stage 2: Performing the diagnosis – Diagnosis is the expression of the "situation of concern" and the reasons why such a state exists.
  - Stage 3: Defining the prognosis outline – Where do we want to be and why?
  - Stage 4: Defining problems – Once the existing and desired states of the situation are understood, there is a need to find out what has been preventing the transformation, i.e., the problem.
  - Stage 5: Deriving notional systems – This is the expression of the system, "if designed, built and operational," it is believed it would eliminate the identified problems.
- Phase 2: Solution design

- Stage 6: Performing conceptual/logical design – In this stage, the solution design is formulated using "systems notions".
- Stage 7: Performing physical design – "Physical design can be considered as the deliberation and selection of *ways and means* of realising the logical design".

Phase 3: Design implementation

- Stage 8: Implementing the design – It is the realisation of the physical design for the "situation of concern".

Evaluation of a method

Evaluation of the three elements of a method, namely, the problem situation, the problem-solving process and the problem solver, is carried out at three stages: before, during and after the method is applied.

2.3.4.1 Formulation of Criteria

Formulation of criteria in these systemic approaches is a rational exercise firmly based on clear philosophical principles. The criteria generated in both the Wieringa's and Jayaratna's approaches are comprehensive and well-organised.

2.3.5 Evaluation Process

The evaluation process used by Wieringa is analytical, whilst NIMSAD promotes the use of action research approach, although analytical techniques can also be deployed.

Despite its relatively narrow focus on modelling techniques, it is clear that the approach used by Wieringa is vastly superior to the random list of criteria previously discussed for many reasons. Wieringa discusses the reasoning process behind the criteria suggested, in terms of why certain models are needed in a method. He uses the same reasoning process to generate the criteria for evaluation, ensuring that the criteria possess a good organisation.

There are certain weaknesses observable in this framework. For example, the treatment of system as essentially a process has some drawbacks. ER model is placed in the column of system decomposition, even though entities do not exhibit the sub-system properties of being "systems in their own right". The claim that

decomposition of system and refinement of interaction can be done independently is also highly questionable. For instance, DFD decomposition requires refinement of data flows and vice versa. Therefore, decomposition of the system and refinement of system interaction seem to be mutual.

When contrasted with other approaches, the NIMSAD framework is unique in many ways. First, it highlights that, practically speaking, effective problem-solving not only depends upon the method used, but also on the situation to which the method is applied and the person who uses the method. As far as the method is concerned, it provides a set of generic phases and steps that any given method should follow and explains why those phases and stages are necessary. Since these system development activities are constructed in very generic terms so that they are applicable to different kinds of methods, or all methods as NIMSAD claims, genericity is both a strength and weakness of the framework. If one is exclusively concerned with, say, CBSD methods, these activities can be of more help if they are more specific. Furthermore, since the framework defines method as a "problem-solving process", there is little scope to incorporate other important parts of modern software development methods such as those based on UML. Therefore, it is fair to say that if the framework is to be used for the technical and theoretical evaluation of a very specific kind of method, it needs to be adapted.

NIMSAD also treats the evaluation of a method as a dynamic activity that is carried out before, during and after the method is applied. This is an original feature of this framework.

2.4 Summary of the findings

As far as the use of empirical approaches to evaluation of CBSD methods is concerned, there are practical difficulties as well as questions regarding its relevance to the nature of inquiry envisaged by this research. Non-empirical approaches are more appropriate in this case. Of the approaches discussed, systemic approaches are the most appropriate because they are based upon sound logical reasoning, and they provide some fairly objective and systematic ways to question the quality of methods. Existing evaluation frameworks such as NIMSAD and Wieringa's provide a good basis for evaluation but they are too generic and need to be synthesised and supplemented with more concrete and detailed criteria as well as mechanisms to

gauge the quality attributes analytically. The next chapter describes such a framework.

Chapter Three

The Proposed Evaluation Approach
The MAP Framework

3.1 Introduction

This chapter expounds a novel evaluation approach proposed by this research, called the MAP framework[4]. MAP stands for Modelling, Architecture and Process. This framework is based upon a number of approaches to understand and evaluate methods, which include systems thinking, NIMSAD and Wieringa's framework. The MAP framework also sheds new light on other important areas of method evaluation such as categorising system models and their correlations, which are not explored by the existing approaches. The proposed framework does not overturn these approaches, but rather synthesises and enhances them by providing more concrete guidelines appropriate for evaluations of CBSD methods. Certain aspects are also applicable to evaluations of other kinds of methodologies, such as Structured and OO methods. Before going into detailed discussion of the evaluation framework, it is worthwhile elucidating some of the key concepts employed: Systems Thinking, System and Information Systems.

3.1.1 Systems Thinking

"Systems Thinking" has its roots in General Systems Theory (GST), which is usually attributed to Ludwig von Bertalanffy (Bertalanffy, 1968) and Ross Ashby (Ashby, 1947). GST suggests that scientific reductionism (Boyd et al, 1991), the idea that a complex problem can be tackled by breaking it down into a set of simpler

[4] A shorter description of this framework has been published in (Bielkowicz and Tun, 2002). An application of the framework to SSADM and UML was presented in (Bielkowicz et al, 2003).

problems to solve, cannot adequately deal with complex biological, social, and other non-physical problems. GST emphasises the importance of interconnectedness of parts of the systems and their complexity. Problems in software development, despite being called an engineering discipline, are *softer* in nature, due to the socio-economic dimension, than those in traditional engineering disciplines such as hardware engineering. Systems thinking is a particular interpretation of the GST applicable to problem-solving in development of information systems. Checkland (1999) defines *systems thinking* as:

> An *epistemology* which, when applied to human activity is based upon the four basic ideas: *emergence*, *hierarchy*, *communication*, and *control* as characteristics of *systems*. When applied to *natural* or *designed systems* the crucial characteristic is the *emergent properties* of the whole.

Epistemology is 'a theory concerning means by which we may have and express knowledge of the world'. *Emergent property* means that systems have properties that are relevant only to the system as a whole, which cannot be attributed to individual components. *Hierarchy* is a concept that is concerned with the relationships between the description of a system/component and a more detailed description of its components. *Communication* means 'the transfer of information' and *control* is the system's ability to exercise regulations (Checkland, 1999).

3.1.2 System

The term system is ubiquitous nowadays. It is used to describe many things from machines such as *computer systems*, to software applications such as *payroll systems*, and from public services such as *transport systems* or *underground systems*, to the interdependency of living things as such an *ecosystem*. All these systems have common properties including the following (Waring, 1996).

- ⇨ Systems have functions
- ⇨ System are made up of a number of parts or components or elements
- ⇨ Components have roles in hierarchical structures
- ⇨ There are means for control and communication
- ⇨ Systems have emergent properties
- ⇨ Systems have boundaries
- ⇨ Systems are affected by their environments

3.1.3 Information Systems

There are a number of definitions of the term 'information system': some of them come from systems thinking perspective and others from methodological background. Jayaratna (1994) provides the following definition, which in a typical systems thinking fashion, emphasises not only the functional aspects of the systems, but also the context in which they operate.

> A system for the most efficient and effective means of identifying the 'real' needs of users, and developing information processing systems for satisfying these needs; ensuring that the resulting information processing systems continue to satisfy changing user needs by the most efficient means of acquiring, storing, processing, disseminating and presenting information; by providing facilities and a learning environment for users and information systems specialists to improve the effectiveness of their decision model; and by supporting operational, control and strategic organisational objectives.

3.1.4 Key Properties of ISs

The MAP framework suggests that there are three key properties of information systems, the proper analysis of which are vital for development of these systems.

INFORMATION: As the name implies, ISs are concerned with information, which is often called 'input and output', or 'signals'. In order to provide useful information, information systems have to carry out tasks such as gathering, storing, manipulating and presenting information.

PROCESS/FUNCTIONALITY: The actions of storing, retrieving, manipulating, presenting information are the functional characteristic of ISs. Some of these actions require the system to communicate with the outside world, such as users and other systems, and to obtain or transmit information through system interactions.

INTERACTION: ISs communicate with the outside world by means of interaction. In a broad sense, interaction means conveying information from a source to a destination.

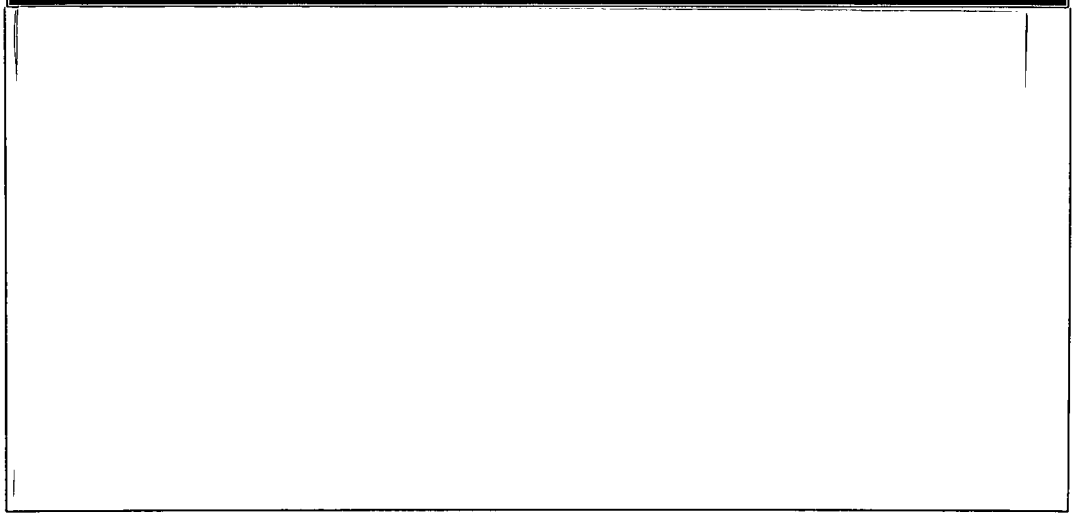
Whilst there are other characteristics associated with ISs, such as emergence, ownership and control, the MAP framework suggests that these are the three most

important characteristics to which developers need to pay paramount attention when analysing and designing ISs. Compared with Wieringa's communication, behaviour and function (Section 2.3.4) these characteristics are less ambiguous.

3.2 Elements of Evaluation in the MAP Framework

At a generic level, this framework agrees with the assertion of NIMSAD that successful evaluation of an SDM requires assessment of not just the method, but also the context in which the it is applied and various personal qualities of the person who applies it. This framework is in line with NIMSAD in saying that the evaluation of the three elements should be done at three stages, before intervention, during intervention and after intervention. Since the NIMSAD framework is designed for the evaluation of a wide range of methods, its definition of method is very generic: 'a problem-solving process'. It is process-focused and for most SDMs, it is rather simplistic. In modern methodologies, such as CBSD methods, specific issues such as modelling techniques are too important to be glossed over. Therefore, the method element of NIMSAD is elaborated by the MAP framework.

Figure 3.1 Elements of Evaluation, Jayaratna (1994)

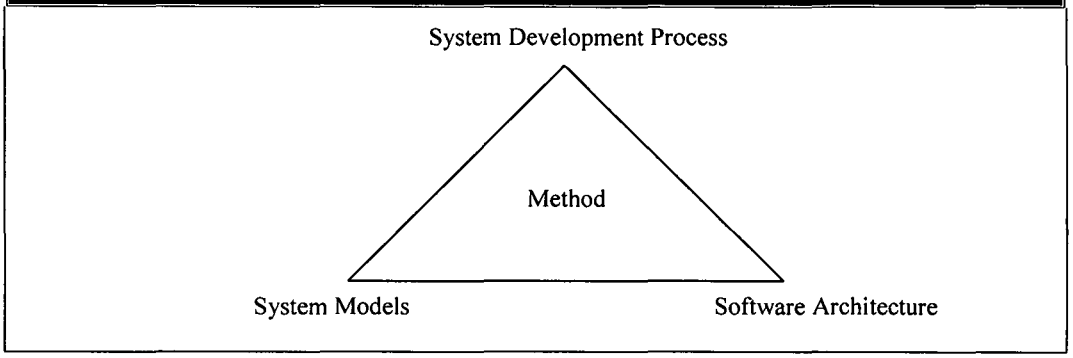


3.3 Three Major Elements of a Method

Based on observations of modern SDMs, the MAP framework suggests that SDMs have three main elements: System Models, System Development Process (SDP) and Software Architecture. These three elements are related to each other, and in a good SDM, the following correlations between the three elements of a method can be observed:

- SDP stages can be expressed in terms of the System Models produced. There are clear correlations between the SDP stages and System Models in terms of when models are created, revised and validated during the development
- Different SDP stages are concerned with different aspects of software architecture, and such correlations are clearly observable.
- System Models, especially design models, have clear correspondence with the software architecture.

Figure 3.2 Elements of Method according to the MAP framework



Therefore, the MAP framework suggests that having these three elements and having clear correlations between them are the highest level criteria in evaluation of a method. From the perspective of a method user, a method cannot be complete or coherent if some of these elements are missing or their correlations unclear. The latitude at which the criteria are applied is high, but these are not too abstract to the point of being unquantifiable. The discussion now will focus on each of the three elements.

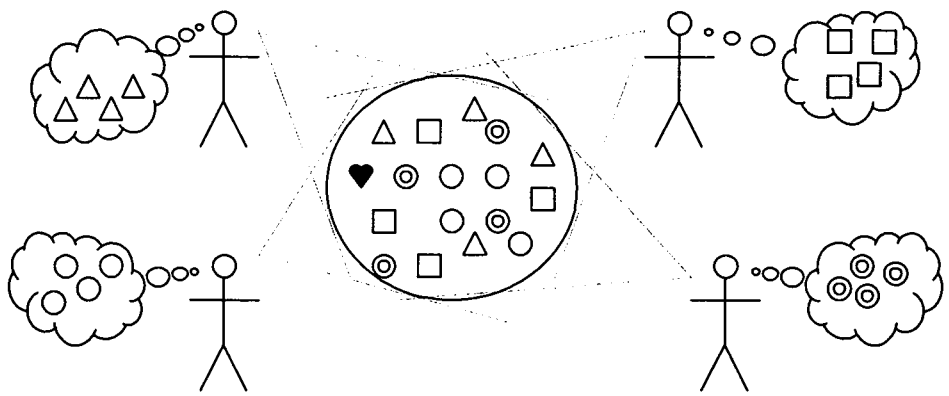
3.4 System Modelling

SDMs provide various system models to project the method-users' reflection of the 'situation of concern'. The projections may be either diagrammatic, formal, informal or a combination of some or all of these. In terms of coverage, the MAP framework suggests that there are two kinds of models: *Global Models* and *Contextual Models*. Global Models are the descriptions of the system in its entirety from a *modelling viewpoint*, and Contextual Models are the descriptions of the context in which a component of the system operates.

3.4.1 Global Models

Due to the complexity of many systems and the limitation of human ability to deal with various complex issues at once, it is difficult in practice to express all important aspects of an entire system in a single model. In fact, it is a general practice for SDMs to use multiple global models, each focusing on one particular aspect of the system and capturing one major characteristic while discriminating others. Therefore, global models tend to give an inherently one-sided view of the system.

Figure 3.3 Model Viewpoints



(Note: There can be elements in a system, which are not covered at all; a heart shape in the diagram.)

3.4.1.1 Modelling Viewpoints

Systems are made up of various kinds of elements (Figure 3.3) and the system, as a whole possesses a range of characteristics, some of which are of vital importance for understanding the way the system functions. For instance, one of the strong characteristics of ISs is the transfer of information between the system and its users across the system boundaries. Therefore, many SDMs can be expected to provide models to capture that particular aspect of the system. In order to produce a model that focuses on one particular characteristic of the system, the modeller takes a standpoint and studies the system from a particular angle, called the modelling viewpoint. Modelling viewpoints are termed differently in methods: for example, SSADM calls them "perspectives", and UML calls them "views" (Goodland and Slater, 1995; Eriksson and Penker, 1998).

Since SDMs provide many viewpoints on a system, the questions of how many modelling viewpoints are needed to get a near-complete description of a system may be raised here. The answer rests with the type of system developers are dealing with;

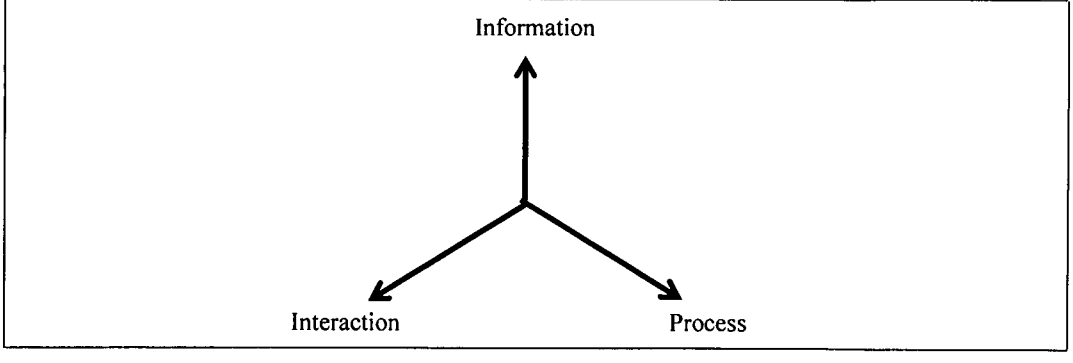
understanding these characteristics is a prerequisite for gaining full comprehension of the system. In this sense, system modelling viewpoints are inseparable from the major characteristics of the system. SDMs should be expected to provide appropriate global models to capture the important characteristics of the type of systems they support. Since this research is particularly concerned with CBSD methods for general ISs, according to the key properties of ISs discussed in Section 3.1.4, SDMs to be evaluated in this research should provide three modelling viewpoints.

If methods to be evaluated are designed for development of a particular kind of IS, such as Real-Time Systems (Burns and Wellings, 2001), which has other important global characteristics such as time. In these cases, additional viewpoints will be needed. It must be noted that all global models have a unique property: they all have the characteristic of *hierarchy*. In other words, global models describe the system at various levels of abstraction or decomposition.

3.4.1.2 IPI Matrix

It is clear from the discussion above that information, process and interaction are important characteristics of ISs. Therefore, it is reasonable to expect SDMs to provide global models to describe these characteristics. In the MAP framework, a polar graph is drawn in which an axis represents each modelling viewpoint necessary for development of a particular type of system. For IS development methods, the polar graph will have three axes for Information, Process and Interaction, named IPI Matrix.

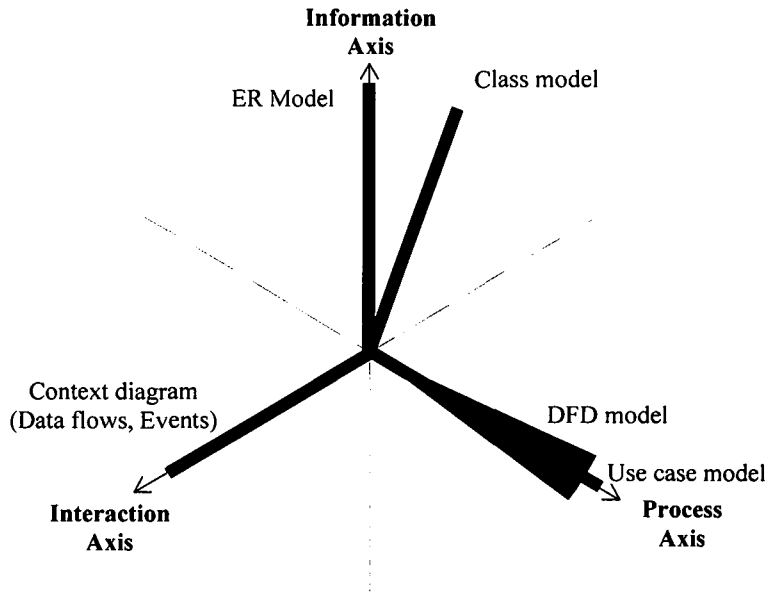
Figure 3.4 IPI Matrix for Global Models of Information Systems



3.4.1.3 Plotting Global Models to IPI Matrix

Each global model from a method is plotted on this matrix by drawing a line along the appropriate axis. According to this framework, an ideal ISD method provides three global models: the process model providing an overall view of the system's functionality, the information model providing a representation of things in the real world that the system it is concerned with, and the interaction model giving a high-level description of the system's communication with the outside world.

Figure 3.5 Global Models of various methods



(Note: ER should be drawn along the information axis; but for the purpose of readability, it is drawn away from it)

ISs have three main characteristics, but it does not follow that they each requires a modelling viewpoint, or that an SDM must provide exactly three global models that fit neatly into this categorisation. For example, some methods may have an information model that contains elements of processes or interactions or both. Such models are accommodated in the matrix by tilting the line towards the appropriate axis or shading an angular area. In the IPI matrix in Figure 3.5, the ER model is drawn very close to the information axis because ER models deal only with the information aspect of the system. On the other hand, the class model has elements of processes, called class operations; hence it is slightly aligned towards the process axis. DFD is essentially a process model but also contains elements of interaction and information. How far a model should be drawn away from an axis is determined by the dotted line which halves the region between the two axes, and represents a

hypothetical model that contains an equal amount of elements from the two neighbouring axes. However, such a model may not have much practical use, since each model should have a clear aim of what it intends to describe.

Certain methods may provide fewer or more than three global models. Again, in these cases, models are aligned appropriately to the axes by first examining the type of elements these models contain. It is worth noting that each global model has a clear purpose: to describe a system from a perspective. As far as ISs are concerned, having more than three main models means that global models contain a large amount of overlap, which is not beneficial since the aims of models can be misleading. Having less than three models means that the aims of the models are ambiguous and their coverage is incomplete. Ideally, all global models should be as close as possible to the axes, ensuring the clarity of purpose.

3.4.1.4 Elements of Global Models

Each global model embodies an aim: for example, the general aim of an information model is to describe the system's information requirements. The main basic units of a process model, an information model and an interaction model are the processing units, logical groups of data items and events (an abstraction of some data flows/messages) respectively. These global models will use various model elements to describe what they aim to represent. Therefore, if a global model is broken down into smaller elements, there will be a certain type of elements that is dominant, reflecting the aim of the global model. A typical information model will contain a disproportionately large number of model elements describing 'basic information units', such as entities. In total, there are three notable kinds of element in a global model. These are:

- **Content/Functional Elements** are essential for the fulfilment of a global model's main aim. For instance, if the global model aims to describe the system's functionality, content/functional elements are then units of processing etc.
- **Structural Elements** show the relationships/dependencies between the content/functional elements, the scope the model, its boundary, its environment etc. They form the scaffolding to support the content/functional

elements. They are also used to add other information such as the boundary/limitations of the model.

- **Overlap Elements** span more than one model since there is a need for traceability between models. Some methods use extra diagrams and other ways of referencing between global models rather than overlap elements.

There can be extra information in a diagram such as diagram title, author, date of creation, version number, and so on. Since such information does not substantially add to the description of the system, it need not be part of a critical evaluation.

3.4.1.5 Hierarchical Nature of Global Models

Due to the size and complexity of systems, detailed descriptions of global models can be overwhelmingly complex. An effective means of managing them is required. Perhaps the most common approach is to break down the system into smaller and comprehensible parts; and describe each part. This is fundamentally a reductionist approach, rejected by systems thinking. The new element brought about by systems thinking, the *emergent property*, requires that, in order not to lose sense of the system at large, the hierarchical structure of the system is used to spell out the role of each part in the system. Therefore, global models are not described in atomic parts; rather their descriptions are expressed at various levels of detail. At the highest level of abstraction, the descriptions will concentrate on the overall structure of the system, and lower level descriptions will contain more detailed information about each component without losing the sense of its place in the overall system. Here an abstract description does not mean an incomplete description; rather it is a description that embodies many detailed descriptions. In terms of building such a hierarchical description of a system, three general approaches are suggested by SDMs: top-down, bottom-up and middle-out.

3.4.1.6 Global models and time dimension

Global models are normally static, i.e. time-independent. This is largely because of the sheer amount of information embedded within them. It is easier to take the time dimension into account when one is concerned with a relatively small portion of the system. This does not mean that a global model cannot be dynamic and a contextual model static. However, if the time dimension is added to a global model, the amount

of information one has to account for will be overwhelming. Static contextual models are very useful for dynamic modelling (see Section 16.3).

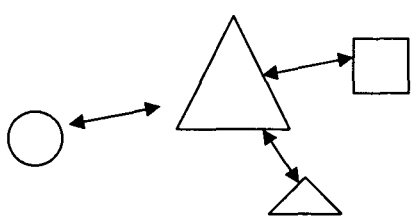
3.4.1.7 Making Sense of Global Models: Integrating Global Models

Clearly global models provide only partial views of the system. There are a number of reasons for consolidating the global models, one of which is to check consistency of these models. In order to perceive the entire system, we need to merge the models together. Now, there is a strong need for a mechanism that will allow one to forge links between global models, paving the way for contextual models. There are two ways to achieve this. The first is to add explicit cross-referencing elements to global models, such as Data Stores in DFD which are directly mapped to entities in the Entity Relationship Diagram. The second is to use a completely separate model that shows only how elements from the two models can be mapped, for example, Entity Access Matrix (EAM); see (Weaver et al, 1998). The first method clearly provides a good starting point for seamless navigation among models; the second is explicit and usually more detailed. It is indeed good to use both. Such relationships between global models are called contextual models.

3.4.2 Contextual Models

A contextual model takes a basic major unit of a global model and describes the system from the perspective of that unit in order to show how the unit of a global model takes part in the running of the system. For instance, a contextual model can show the system from the point of view of a unit of processing in terms of how it relates to information unit(s) and interaction unit(s) to perform a specific task. Global models are outside views of the system in its entirety; contextual models are inside views of the system limited to a single element.

Figure 3.6 Contextual Models show how components from Global Models correlate

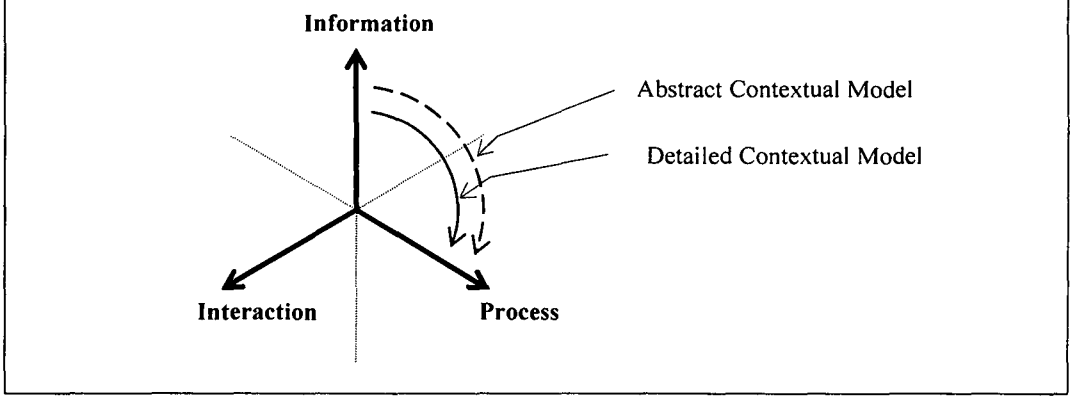


Contextual models tend to transcend limits of global models in the sense that they may contain information from all global models. For example, when describing the system from a processing unit's point of view, analysts can freely mention how it interacts with elements of other global models and vice versa. Therefore, overlaps between the global models are crucial for contextual modelling.

3.4.2.1 Two Kinds of Contextual Models

In the IPI matrix, contextual models are shown as arrow-headed arches. The MAP framework distinguishes between two kinds of contextual models. The first is the abstract contextual model that shows in generic terms, following the direction of the arrow, how one element from a global model relates to element(s) from another global model. The second is a detailed contextual diagram. The former is represented by the dotted arrow and the latter, a thick arrow.

Figure 3.7 Contextual Models show abstract and detailed relationships between elements of global models



Though contextual diagrams are classified in this way, there is no need to have separate diagrams for each of the contextual models. A simple table such as Entity/Event Matrix in SSADM shows abstract contextual relationships between all entities and events in the system. Detailed contextual models expand on these abstract relationships and provide greater analyses of the correlations. For example, an ELH in SSADM expands on the effects of events on an entity shown in the EAM.

3.4.2.2 Time Dimension

Since contextual models, especially detailed contextual diagrams, describe the system from the individual component's perspective, their descriptions of the system tend to be detailed and therefore they normally have a time dimension. For example,

the state transition diagram for an object will clearly have a time dimension. Although in theory, it is possible to draw a state diagram for the entire system, the diagrams are better suited for individual objects because the sheer amount of detail analysts have to deal with is much more manageable on the scale of an individual element. Contextual models are often known as dynamic models because of their time dimension.

3.4.2.3 The relationship between global models and contextual models

Global models show the system as a whole, and contextual models focus on individual elements in these global models. In one sense, contextual models bind global models together. The depth of information and dynamism of contextual models presents a good opportunity to check the validity and consistency of the global models.

The discussion so far has outlined what can be called a *theory of system development method*, in the sense of the theory of mind, as assumed by the MAP framework. The following criteria and evaluation process are derived from this theory.

3.4.3 Evaluation of System Modelling

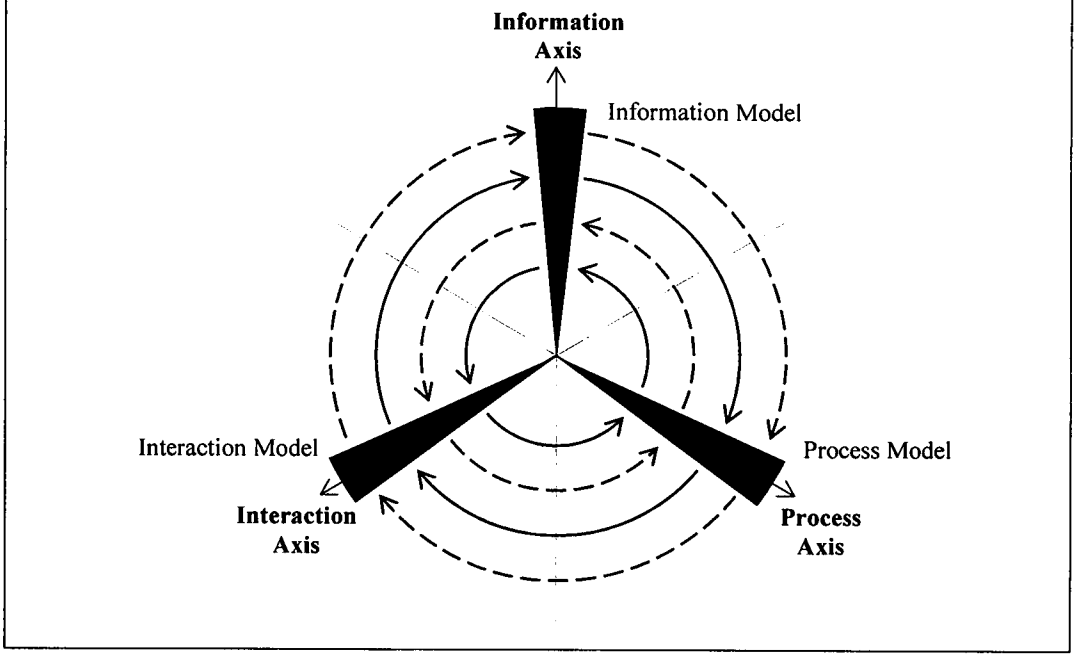
Evaluation of modelling is primarily concerned with the coverage of all models as a whole in describing various important aspects of the system, and examining how closely they are held together. According to this theory of system development method, an ideal method will have:

- three global models describing the information, process and interaction characteristics of the system
- six abstract contextual diagrams
- six detailed contextual diagrams

General alignment of a model in the IPI matrix, the rigour of the modelling techniques, and its consistency with other models will indicate the overall quality of a model, according to the MAP framework. Furthermore, models should also have clear correspondence with the stages at which they are developed, revised, and finalised and with the aspects of software architecture they correspond with. Having this many models in a method may be considered bureaucratic and impractical

(Hares, 1994). However, this defines the best possible set of models required to describe a system, against which a method can be evaluated.

Figure 3.8 An Ideal Set of Models Envisaged by the MAP framework



The system modelling evaluation criteria of the MAP framework are divided into two categories: criteria for evaluation of techniques used by global models and criteria for evaluation of techniques used by contextual models. It can be argued that "consistency" is perhaps the single most important quality of requirement specifications. Consistency has two important dimensions: external consistency, a specification reflecting users' real requirements, and internal consistency, the specification not contradicting itself or other specifications. The MAP framework suggests specific criteria for evaluation of techniques used by global and contextual models in order to measure how rigorously external and internal consistencies of system models are ensured by SDMs.

3.4.3.1 Criteria for Evaluation of Techniques used by Global Models

These criteria refer to specific quality attributes of individual global models. The criteria and the evaluation process are based on the rigorous approach described in our paper on a comparison of data requirements specification techniques (Bielkowicz and Tun, 2001).

The evaluation approach suggests that quality of a global model/specification, in that case a data requirements specification, can be evaluated at three main levels: Description Level, Semantic Level and Contextual Level. The first is a superficial level at which a common understanding is thought to have established if a shared vocabulary is used. The second level is more meaningful, whilst the third applies to consistency between models. For the first level, there are two main quality characteristics: completeness and minimality, which ensure that the specification matches squarely with what is required. At the semantic level, the quality characteristics correctness and non-redundancy are identified; these ensure that there is no difference between the meaning of the languages used by the developers and the users. For the contextual level, only the inter-model consistency criterion is suggested in the paper, which is subsumed in this framework by the criteria for evaluation of contextual models. These criteria are then applied to all elements in the model in all levels of abstraction. In data requirements specifications, the main elements are: data groups (entities/classes), attributes, and relationships between entities or classes.

Table 3-1 Criteria for Evaluation of an Information Model

Level	Characteristic	Element	Criterion
Description	Completeness	Entity/Class	Completeness of entities/classes
Description	Completeness	Attribute	Completeness of attributes
Description	Completeness	Relationships	Completeness of relationships
Description	Minimality	Entity/Class	Minimality of entities/classes
Description	Minimality	Attribute	Minimality of attributes
Description	Minimality	Relationships	Minimality of relation-ships
Semantic	Correctness	Entity/Class	Correctness of entities/classes
Semantic	Correctness	Attribute	Correctness of attributes
Semantic	Correctness	Relationships	Correctness of relationships
Semantic	Non-redundancy	Entity/Class	Non-redundancy of entities/classes
Semantic	Non-redundancy	Attribute	Non-redundancy of attributes
Semantic	Non-redundancy	Relationships	Non-redundancy of relationships
Contextual	Consistency	Related specifications	Consistency of the specifications

The levels of description and characteristics are essentially the same for all global models. Since global models contain different sets of elements, specific criteria for evaluation of a global modelling technique are generated by applying the four characteristics to each major element in that model.

When evaluating the specification techniques, the guidelines are first summarised and then their rigour is measured using a simple scale where:

- 0 means no guidelines are provided
- 1 means some guidelines provided but they are weak or implicit
- 2 means guidelines are clear and strong

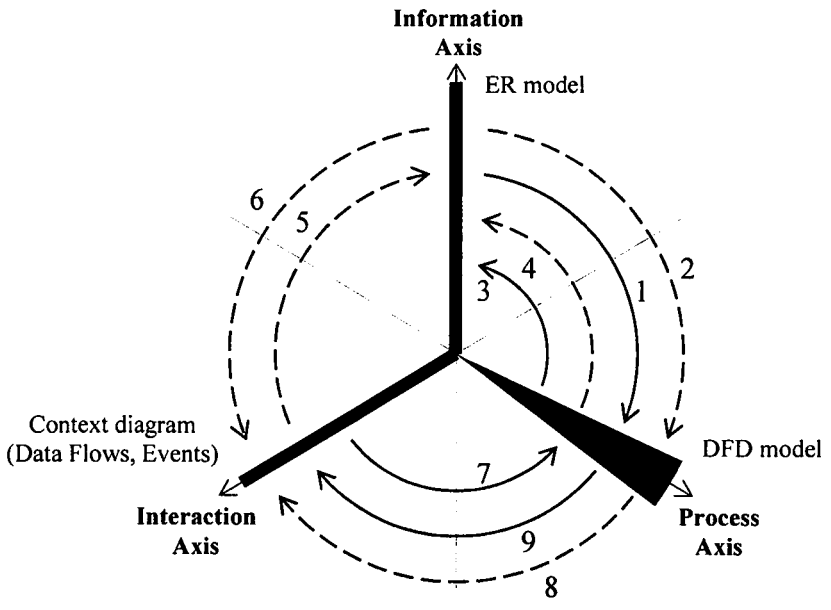
The grading for each criterion is then added up to give a figure indicating the total rigour of the modelling technique for a global model. Relative strength of a modelling technique can be calculated using the following formula:

Strength = Round (Total Rigour / (Number of Criteria * Maximum Rigour Grade) * 100)

Strength therefore means the extent to which a modelling technique achieves the highest rigour score.

3.4.3.2 Criteria for Evaluation of Techniques used by Contextual Models

Figure 3.9 The IPI Matrix showing all major models of SSADM



1. Entity Life History (including operations)
2. A column in the Entity Event Matrix
3. Effect Correspondence Diagram (including operations)
4. A row in the Entity Event Matrix
5. A DFD fragment showing how a data flow affects a number of storages via process/processes
6. A DFD fragment showing how a storage is created/updated/deleted/read by a number of data flows via processes
7. Explicit in DFD
8. I/O Structures
9. Also explicit in DFD

Contextual models will largely make use of elements already used in global models, and therefore, the main issue here is to assess how linkages between elements of global models are identified. The criteria for evaluation of a contextual modelling technique are generated by applying description level characteristics – namely, Completeness and Minimality – to major elements of the contextual model.

3.4.3.3 Evaluation Process

For each global model provided by a method, draw a line in the IPI Matrix as shown in Figure 3.9. The position of the line depends upon the aim of the model. Then discover the contextual models, and draw appropriate arches on the matrix. Then from the matrix, ascertain the coverage of analysis models provided by the method, i.e., if there are complete circles, ideally four of them, models have a good coverage. Then evaluate the modelling technique of each global and contextual model using the criteria discussed in previous two sections.

3.5 System Development Process (SDP)

The MAP framework considers that there are two related issues in a SDP model: Development Activities that developers need to carry out in a typical project and a Control Mechanism, which applies regulations to the development activities. For example, the classical "waterfall model" suggests that the requirements definition, system and software design, implementation and unit testing, integration and system testing, and operation and maintenance are the main stages of system development activities. The control mechanism it uses indicates that those activities should be carried out in that sequential order (Sommerville, 2004).

3.5.1 Evaluation of System Development Process (SDP)

Evaluation of a SDP model in the MAP framework is divided into evaluation of its development activities and control mechanism.

3.5.1.1 Criteria for Evaluation of Development Activities

The NIMSAD framework provides a set of logical steps and stages that any system development project should take. These steps and stages serves as a set of criteria against which a given system development process can be evaluated. As noted in Section 2.3.4, these steps and stages are too generic for a critical evaluation of very

specific kinds of method, in this case, CBSD methods. The MAP framework remedies this shortcoming by synthesising generic development activities with considerations for unique features of CBSD.

A number of component-based SDPs have been examined in this research including those put forward by authors of the evaluated CBSD methods and many others, such as Cheesman and Daniels (2001) and Pressman (2005). It is clear from these SDP models and general commentary of CBSD that the defining theme of CBSD is to create applications from developed components by reusing them, and its key development activities include creating/acquiring components and assembling them. By combining specific features of CBSD with NIMSAD, the MAP framework proposes that a component-based SDP should entail the following development stages:

- Feasibility Analysis – Before the technical, financial, technological, methodological and component feasibilities of the project are established, an understanding of the problem situation needs to be acquired through basic business and functionality analysis. This stage is similar to the 'understanding the situation of concern' in NIMSAD.
- Business Modelling – Business Modelling usually involves an expression of the current situation and the new situation. Analysis of the current situation and the gap between the current and required situations constitutes the "performing the diagnosis" stage of NIMSAD, while the expression of the desired situation constitutes the "defining the prognosis outline" stage of NIMSAD.
- Requirements Analysis – The requirements specification produced in this stage defines what the required system should do. This step corresponds with the "defining problems" of NIMSAD.
- System Analysis – NIMSAD seems to treat the development of notational models of the system as a separate step, but most methods suggest that the models are developed in the development stages. Arguably, most of these models are produced during system analysis.
- Logical Architecture – Architectural design deals with issues such as decomposition of the system into logical components and an analysis of their dependencies. The design is essentially logical because it is

independent of the implementation technology. NIMSAD calls this "performing logical design".

- Physical Design – This design is more detailed and implementation-oriented, and is generally called physical design.
- Component Search – This is a CBSD-specific task dealing with the search for existing components and applications that are reusable in the project.
- Component Certification – Identified components need to be verified and validated against their specifications before they can be integrated into the application.
- Component Implementation – Components for which there are no reusable candidates need to be implemented.
- Application Assembly – Components are put together to create the application.
- System Testing – Testing of the system/application as a whole to ensure that it satisfies both functional and non-functional requirements.
- System Delivery – Hand-over the application from the developers to the users.

These stages serve as a set of criteria for evaluating coverage of CBSD processes in the MAP framework. Regarding these criteria, there are two important points to note here.

First, these stages are by no means exhaustive; this is not a super-process that encompasses all imaginable tasks developers will come across in a development project. It is, in fact, the least that can be expected of a CBSD process, the lowest common denominator. Methods may provide additional detailed activities.

Second, these criteria do not state whether some stages are more important than others. Depending on the principles of a method, emphasis may be placed upon different stages; for example, a CBSD method, based on the principle of rigorous modelling prior to the development of software, will play up the importance of the early stages, while another method based on the principle of rapid development, such as the agile development approach, will play down the importance of these stages. On the whole, the MAP system development process criteria do not promote one approach over others. The framework disagrees with approaches that call for a

complete demotion of system modelling such as eXtreme Programming (Beck, 1999).

3.5.1.2 Criteria for Evaluation of Control Mechanisms

In addition to providing good coverage for development activities, a chosen SDP should have an appropriate control mechanism suitable for the project. Since different control mechanisms are suitable for different types of project, the main issue here is not about deciding whether one control mechanism is better than another, but rather about choosing a control mechanism that a particular application development needs.

Figure 3.10 Different Types of Applications and Suitable Control Mechanisms

Type of Application	Suitable Control Mechanism
Requirements for the application are stable, clear and can be well-defined A high level of accuracy required, e.g. applications using scientific algorithms	Linear, Spiral
Requirements for the application change frequently	Iterative, Incremental, Prototyping
Requirements are unknown or uncertain; the situation is new and needs innovation and learning	Exploratory Prototyping
The environment changes in reaction to the system under development	Evolutionary Prototyping

Therefore, the MAP framework suggests that a control mechanism can only be evaluated in terms of its suitability for a given project. Based on the 'organisational environments' and their characteristics, explained by Land (1998), the MAP framework provides the following correlations between different types of applications and suitable control mechanisms, which can help developers make a choice. According to Land (1998), most business applications have changing requirements, and therefore, component-based development of these applications requires control mechanisms that are iterative, incremental, and prototype-based. Development of other types of applications, such as real-time applications, requires rigidity of the linear control mechanism.

3.5.1.3 Evaluation Process

Evaluate the coverage of a given SDP by comparing it against the criteria given in Section 3.5.1.1, and determine the suitability of the control mechanism for the project by assessing the nature of application as discussed in the previous section.

3.6 Software Architecture

Since CBSD encourages creation of applications by assembling components, possibly from different sources, the question of how to determine the best design for the application is crucial. Therefore, the MAP framework expects CBSD methods to provide guidelines for producing a good architectural design. The term "software architecture" has been defined in many ways (SEIa, n. d.); Garlan and Shaw (1993, 1996), for example, provide the following simple definition, which is reasonably good for discussions in this thesis.

> The architecture of a software system defines that system in terms of computational components and connections among those components.

3.6.1 Two kinds of architecture

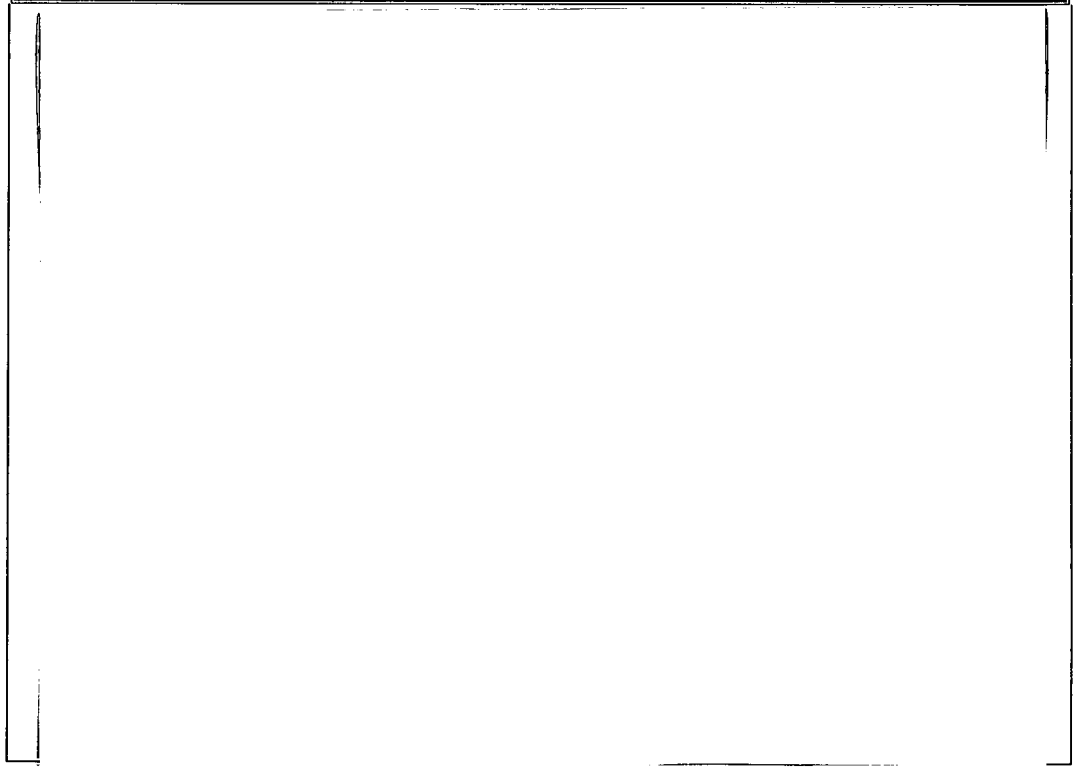
The MAP framework recognises the need for the separation of architecture into logical architecture and physical architecture, as suggested by NIMSAD.

3.6.1.1 Logical Software Architecture

The primary focus of this logical software architecture is to help identify high-grained components and structure of the system without reference to implementation technologies. Logical architectural analysis, performed at the earlier stage of development with a view to identifying and specifying major components of the system, is important because this model will lay the foundation for later development. Many strategic architectural decisions such as security, reliability, and functionality of each component are taken here. To assist with this activity, many methods offer a "reference architecture", which is a "generalised architecture of several end systems that share one or more common domains" (Gallagher, 2000). These reference architectures spell out, in application-nonspecific terms, the main components of systems and their relationships. For instance, Cheesman and Daniels (2001) propose a four-layered architecture, which is divided into two: a client part and a server part. Components in the User Interface and User Dialogue layers of the client part will deal with the user interaction aspect of the system, while components in the System and Business Services layers will encapsulate the business logic and database. When developing an application using this architecture, analysts will then

know what kind of component they should be looking for in each layer, hence the name, "reference architecture".

Figure 3.11 A Reference Architecture by Cheesman and Daniels (2001)



3.6.1.2 Physical Architecture

Physical architecture is concerned with lower-level, or component level, implementation-dependent design of components. The logical architecture is concerned with issues external to components, such as communication, whilst the physical design is concerned with issues internal to components. Again, CBSD methods are expected to provide guidelines on producing good internal designs.

3.6.2 Evaluation of Software Architecture

Architecture is most receptive to the use of empirical evaluation approaches. Much research has already been done in this area, notably by researchers at the Carnegie Mellon Software Engineering Institute (SEIb, n. d.). The MAP framework does not propose any new evaluation approach for software architecture; the discussions here will only point out how some of the existing evaluation approaches can be used within the context of the MAP framework.

3.6.2.1 Evaluation of Logical Architecture

Since reference architectures cater for different types of application, the issue here is how to select a reference architecture that is suitable for a given application. Traditionally, software architectures are evaluated using a narrow set of criteria, such as performance and reliability (Bass et al, 1998; Kazman et al, 1996; Smith and Williams 1993). However, it is well known that there are many important quality attributes in software applications, and that they often conflict; for example, efficiency versus usability. Choosing the right architecture for an application is, therefore, a balancing act. One main contribution of researchers at the Software Engineering Institute is an architecture evaluation approach that helps evaluators make a conscious decision about the quality compromise they want to achieve. The evaluation approach is called that Architecture Tradeoff Analysis Method (ATAM). The ATAM and case studies for its applications have been described in a number of publications such as (Kazman et al, 1998), (Gallagher, 2000) and (ATAM, n. d.). Kazman et al (1998) describe the ATAM as a six-step process in which requirements and quality attributes are identified, and competing architectural designs are produced and evaluated. Steps of the ATAM are:

- Step 1: Collect Scenarios – usage scenarios are elicited, and requirements, constraints, and environment details are identified from stakeholders.
- Step 2: Collect Requirements/Constraints/Environment – "attribute-based" requirements are identified; Step 1 and Step 2 are interchangeable.
- Step 3: Describe Architectural Views – analysis of architectural properties is captured in "architectural views," such as module view, process view, dataflow view, class view, and so on.
- Step 4: Attribute-Specific Analyses – each quality attribute is analysed in isolation with respect to each architecture.
- Step 5: Identify Sensitivities – attribute values that are significantly affected by deign change to a particular architectural element, known as "sensitivity points," are identified.
- Step 6: Identify Tradefoffs – architectural elements with multiple sensitivity points are identified and interactions between the attribute values analysed.

The evaluation process is iterative, part social and part technical. The MAP framework recommends the use of the ATAM for evaluation of logical architecture.

3.6.2.2 Evaluation of Physical Architecture

For comparative assessment of competing architecting, Shaw (1994) has suggested a three step evaluation approach involving "model problems", which is further explained in other publications such as (Shaw et al, n. d.; Model Problem, n. d.).

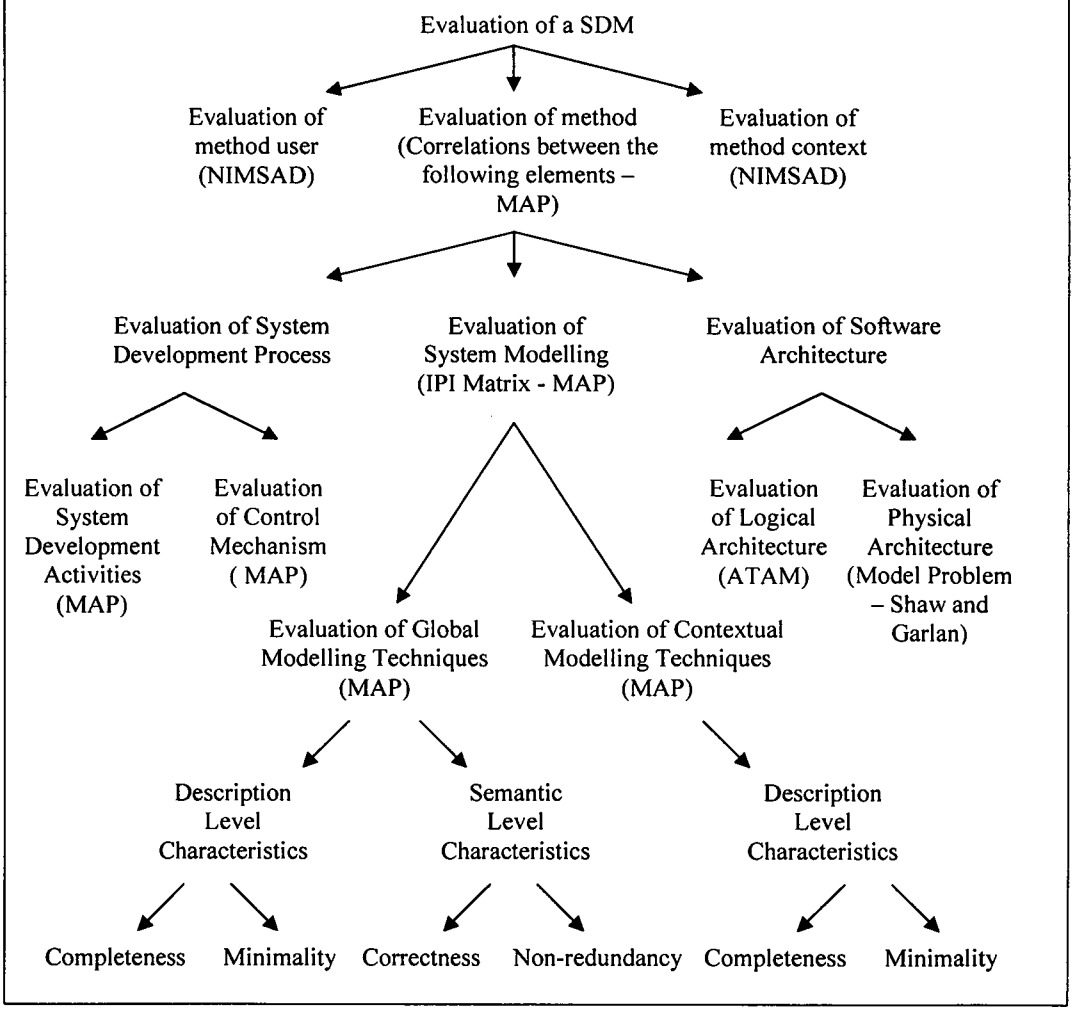
- Step 1: First choose a type problem – a kind of multi-purposed case-study scenario "that set[s] a minimum standard of capability for new participant" is chosen. Shaw et al (n. d.) explored 11 problems ranging from Automated Teller Machine (ATM) to Library system.
- Step 2: Develop a design model that conforms to the architecture – following the techniques and modelling languages provided by the methods, designers will produce architectural designs of the chosen model problem. However, Shaw acknowledges that a good design is to some degree subjective.
- Step 3: Compare the resultant designs by applying a set of criteria – Shaw has used the following criteria: generic criteria include "Separation of concerns and locality," "Perspicuity of design," "Ability to analyse and check the design" and "Abstraction power". For application-specific criteria (for the Cruise-control problem), Shaw assesses "Safety" and "Integration with the vehicle."

Since this is low-level design, design metrics such as (Washizaki et al, 2002; Washizaki et al, 2003) can also be used.

Due to limitation of scope and time, this research will not explore on empirical evaluation of architecture; rather, it will focus on important methodological issues, namely, what components are, and how they are identified and validated in various CBSD methods.

3.7 Summary

Figure 3.12 Structure of the MAP Framework



In the MAP framework, criteria for evaluation of a method are organised hierarchically as shown in Figure 3.12. At the highest level, the evaluation covers three key elements – method user, method and the method context – as suggested by NIMSAD. The MAP framework focuses on the evaluation of the method which it suggests has three elements: SDP, System Modelling and Software Architecture. Evaluation of a method starts with an analysis of the correlations between these three elements. When evaluating SDP, the MAP framework divides the evaluation criteria into two groups. For evaluation of development activities of an SDP, the MAP framework provides a set of development activities using which the coverage of the SDP can be determined. For evaluation of the suitability of the control mechanism of the SDP for a particular application development, the framework

identifies correlations between types of application and suitable control mechanisms. As far as modelling is concerned, an IPI Matrix is produced showing the relationships between major models of a method. Modelling technique for each global model is evaluated using evaluation criteria generated by applying the four quality characteristics of a global model – completeness, minimality, correctness, non-redundancy – to model elements used by the model. Criteria for evaluation of contextual models are generated by applying completeness and minimality to model elements used by the contextual model. Strength of a modelling technique is measured by grading the rigour for each criterion. Software architecture is evaluated by using the ATAM approach and the "model problem" approach.

Chapter Four

Evaluation of Existing CBSD Methods

Applying the MAP framework

4.1 Introduction

This chapter presents discussions on the evaluation of the following CBSD methods, in chronological order of their publication, using the MAP framework.

- Reuse-driven Software Engineering (RSE) by Jacobson et al (1997)
- SELECT Perspective by Allen and Frost (1998)
- Catalysis by D'Souza and Wills (1999)
- Component-based Product Line Engineering with UML or KobrA by Atkinson et al (2002)

The initial plan to present the evaluation of these methods was to summarise the key features of each method before going on to evaluate them with reference to the points highlighted in the summary. This format was used in our published paper on the evaluation of data requirements specification techniques (Bielkowicz and Tun, 2001). However, since that style of presentation would make this document extremely long, a decision has been taken to present an analytical summary of existing CBSD methods in journal paper format and attach it to Appendix II. The paper has been prepared with an intention for later publication.

Evaluation of these methods in the following sections is organised as follows. For each method, correlations between the three elements of the method, namely Software Architecture, System Modelling and System Development Process, are assessed. This is the top level criterion. Each of the elements is then evaluated individually. For evaluation of models, an IPI matrix is produced showing the global and contextual models of the method, and conclusions regarding the coverage and rigour of crosschecking are derived. The modelling technique for each model is then evaluated using a detailed set of criteria formulated according to the discussions

provided in Sections 3.4.3.1 and 3.4.3.2. Rigour of the modelling technique is also graded. For evaluation of software architecture, the nature of components, their identification and validation techniques are discussed. Finally, the development process of the method is evaluated according to the criteria discussed in Section 3.5. This chapter concludes with a report on the results of an experiment carried out to test whether independent practitioners of the MAP framework can produce similar IPI matrixes for two well-known methods.

4.2 Evaluation of Reuse-driven Software Engineering (RSE)

A summary of RSE is provided in Section 2 of Appendix II.

4.2.1 Correlations Between the Three Elements of a Method

RSE provides all three elements of a method as required by the MAP framework (Section 3.3), and the following correlations between these elements can be observed.

- RSE regards the development process as a series of model development. There are explicit mappings between the system development stages and system modelling (see Figure 2-2 of Appendix II).
- The relationships between the system development process and software architecture are also clear because one of the main sub-processes is called Application Family Engineering (AFE), which specifically deals with the architectural issues (see Section 2.4 of Appendix II).
- The links between the modelling and software architecture are implicit because modelling in AFE sub-process largely relates to the software architecture.

At this latitude, some cohesion between the three elements of the method can be noted. Each of these elements will now be evaluated individually.

4.2.2 Evaluation of System Modelling

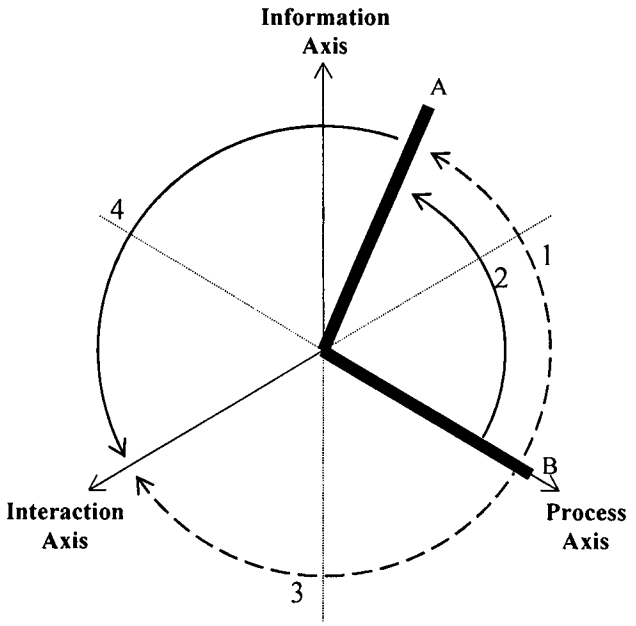
Figure 4.1 shows the IPI Matrix produced for RSE models according to discussions given in Section 3.4.1.

As far as the coverage is concerned, the following conclusions can be drawn.

- Out of three possible global models, RSE provides two; there is no model in RSE that explicitly deals with the interaction between the system and the

user at the global level. Therefore, though both state diagram and use case descriptions are unified through the concept of stimuli, there is no global diagram expressing the input/output interactions between actors and the system.

Figure 4.1 IPI Matrix showing RSE models



A. Class Model
B. Use Case Model
1. Robust Analysis showing objects of different types contribute to a use case
2. Interaction Diagram showing how messages are passed between objects to realise a use case
3. Step-by-step descriptions of the use cases indicating stimuli of a use case
4. State Model showing how states of a class make it respond differently to stimuli

- Out of six possible abstract contextual models, this method provides only two. This means a serious lack of abstract contextual models.
- Out of six possible detailed contextual models, this method provides only two, of which RSE only emphasises the interaction diagram. This is also a major shortcoming of RSE.

4.2.2.1 Evaluation of Use Case Modelling

Use Case Modelling of RSE uses the concepts Use Case, Actor, (Actor-Use Case) Association and (Use Case-Use Case) Relationships. The criteria in the first column of the table are generated according to the discussion in Section 3.4.3.1. The second column summarises the modelling guidelines relevant for each criterion and the third

column indicates the rigour of those guidelines. Similar tables will be used to show evaluation of other modelling techniques in this chapter.

Authors of RSE have invented "superordinate" use cases and actors to capture requirements at a high level of abstraction, such as the requirement for an entire application. Due to the word limitation of this thesis, they will not be distinguished here. Modelling guidelines are partly given in the RSE book, with numerous references to OOSE. This evaluation covers guidelines provided in both books.

Table 4-1 Evaluation of RSE Use Case Modelling

Criterion	Modelling Guidelines	Rigour
(a) Completeness of use cases	By analysing the ways various actors use or interact with the system, use cases are identified (p. 159, OOSE). RSE suggests a number of inputs to this process such as customers, business models and existing components. It generally regards identification of use cases as obvious and no checks are provided to ensure completeness of use cases. Although domain object model and use case model can be crosschecked systematically, no guidelines are provided for this.	1
(b) Completeness of actors	Actors are identified by analysing roles of those who (will) use the system. Although it is straightforward to identify actors, an SDM needs to provide checks to ensure the completeness, and RSE does not have such checks.	1
(c) Completeness of associations	These associations are usually derived from the analysis of use cases. Since the meaning of these associations is rather vague, it is difficult to ensure that all relevant associations have been identified. For example, if a reader borrows book through a library assistant, it is difficult to see how the association(s) should be indicated.	1
(d) Completeness of use case relationship	OOSE suggests potential cases in which use case relationships such as <<extends>> can be used. For example, extension is used to 'model optional parts of use cases' (p. 165, OOSE). There are no clear conditions for when and when not to use these relationships.	1
(e) Minimality of use cases	Participation of users in the development of use cases and the problem domain objects model can help remove unnecessary use cases, actors, and their relationships. This elimination of extraneous model elements can be done much more methodically; however, RSE does not provide guidelines in this respect.	1
(f) Minimality of actors	See (e).	1
(g) Minimality of associations	See (e).	1
(h) Minimality of use case relationship	OOSE does not indicate when use case relationships become unnecessary and should be eliminated.	0
(i) Correctness of use cases	RSE relies on user participation, business models and existing components to ensure that the requirements model is correct. Specific mechanisms are however	1

	lacking.	
(j) Correctness of actors	See (h).	1
(k) Correctness of associations	See (h).	1
(l) Correctness of use case relationship	OOSE provides some arguments for and against using low-grained use cases, meaning that there are cases where many use case relationships should be used and there are cases where few use case relationships should be used. Authors of OOSE are in favour of the former, they do not state firmly when (p. 174, OOSE).	1
(m) Non-redundancy of use cases	The usage of use case relationships such as <<uses>> can help remove overlaps between use cases.	2
(n) Non-redundancy of actors	Generalisation of actors can help remove redundancy between actors.	2
(o) Non-redundancy of associations	No guidelines are provided.	0
(p) Non-redundancy of use case relationship	No guidelines are provided.	0
	Total	15
	Strength of the RSE use case modelling technique	47

4.2.2.2 Evaluation of Class Modelling

The main elements of class modelling are Class, Attributes, Operations, Inheritance and Associative Relationships (Association, Aggregation, Composition and their cardinalities). By applying the quality characteristics discussed in Section 3.4.3.1 to these elements, the criteria in the first column of the following table are generated.

Table 4-2 Evaluation of RSE Class Modelling

Criterion	Modelling Guidelines	Rigour
(a) Completeness of classes	In OOSE, first a noun analysis of business concepts is carried out to identify problem domain objects. Then in the analysis model, entity and interface objects are identified from use case descriptions. Exactly how and when control objects should be used is rather unclear. RSE suggests that there are additional sources of information such as business model, existing components etc.	1
(b) Completeness of attributes	OOSE is ambiguous about this completeness. It suggests that it is often difficult to decide how to model certain information; what is important is how the information is used: "Information that is handled separately should be modelled as an entity object, whereas information that is strongly coupled to other information and never used by itself should be made into an attribute of an entity object." (p. 185-188, OOSE)	1
(c) Completeness of operations	Since entity objects can be manipulated only through operations, all access to these objects must be made through operations. Descriptions of use cases provide the source of information for operations (p. 188, OOSE). Later, interaction diagrams are used to analyse interactions between objects, where operations are	2

	further identified.	
(d) Completeness of inheritance relationships	OOSE only discusses what inheritance relationships mean; it is left to the analysts to decide when to use them.	1
(e) Completeness of associative relationships	These relationships are added when instances of the classes communicate by invoking operations of each other.	2
(f) Minimality of classes	Only those objects that participate in the use cases are included in the analysis model.	2
(g) Minimality of attributes	No guidelines are provided.	0
(h) Minimality of operations	It is determined implicitly; operations of classes that do not contribute to use cases are unnecessary.	1
(i) Minimality of inheritance relationships	No guidelines are provided.	0
(j) Minimality of associative relationships	Implicit from (e).	2
(k) Correctness of classes	OOSE emphasises that the participation of users in the development of the use case model, user interface descriptions and the problem domain model can help here.	1
(l) Correctness of attributes	No guidelines are provided.	0
(m) Correctness of operations	Guidelines for determining the nature of operations are typically vague. OOSE suggests that there are two extremes here: in one extreme, only "set" and "get" operations are allocated, and in the other extreme "whole course of events" is included in operations. "As always," it suggests, "the right thing is the middle course between these extremes."	1
(n) Correctness of inheritance relationships	No guidelines are provided.	0
(o) Correctness of associative relationships	It is unclear how this correctness is ensured in RSE.	0
(p) Non-redundancy of classes	As in (r), OOSE suggests 'homogenisation' of classes, so that classes that are not related by inheritance do not provide similar functionality. (p. 243)	2
(q) Non-redundancy of attributes	No guidelines are provided.	0
(r) Non-redundancy of operations	OOSE talks about 'homogenisation' of stimuli for operations, by which, its authors mean minimising the set of operations required from a class. (p. 228, OOSE)	2
(s) Non-redundancy of inheritance relationships	No guidelines are provided.	0
(t) Non-redundancy of associative relationships	No guidelines are provided.	0

	Total	18
	Strength of the RSE class modelling technique	45

4.2.2.3 Evaluation of Interaction Diagram

Interaction diagrams are used to show how messages are passed between objects when a use case is executed. The main concepts are: participating Objects, Operations and the Flow of the messages passed.

Table 4-3 Evaluation of RSE Interaction Diagram

Criterion	Modelling Guidelines	Rigour
(a) Completeness of participating objects	Participating objects come directly from the robust analysis, in which interface, control and entity objects of a use case are identified. Use cases are described in steps and operations are allocated to carry out the tasks.	2
(b) Completeness of operations	Operations are identified from the stimuli that the system receives from the actor(s). They are identified by looking at how the actor(s) interact with the system through the interface.	2
(c) Completeness of flows	This is dependent on the description of the use cases.	1
(d) Minimality of participating objects	RSE and OOSE do not suggest how to identify and eliminate unnecessary participating objects.	0
(e) Minimality of messages	Messages are minimised by rationalising the stimuli. (p 220-221, OOSE)	2
(f) Minimality of flows	Not provided.	0
	Total	7
	Strength of RSE Interaction Diagram modelling technique	58

4.2.2.4 Evaluation of State Diagram

RSE only mentions state diagrams and the discussion refers to the technique discussed in OOSE. The main concepts used are States and Event/Transition.

Table 4-4 Evaluation of RSE State Diagram

Criterion	Modelling Guidelines	Rigour
(a) Completeness of states	In OOSE, stimuli and operations from interaction diagrams are crucial for determining the states and transition of the objects. It is not clear how exactly states are identified.	1

(b) Completeness of events/transitions	Events/Transitions are identified rather intuitively.	1
(c) Minimality of states	No guidelines are provided.	0
(d) Minimality of events/transitions	Implicit from 'homogenisation' of stimuli.	1
	Total	3
	Strength of the RSE State Diagram modelling technique	38

4.2.3 Evaluation of Architecture

RSE Architecture is summarised in Section 1.1 of Appendix II. The RSE reference architecture is intended for design of a set of related applications, called Application Family, which share a number of common features. The applications it envisages are very large, complex, and geographically dispersed. They are suitable for large business applications.

4.2.3.1 Definition of Component

RSE defines the term "component" as "anything specifically engineered to be reusable". Therefore, any development artefact, whether it is a class, a use case, a fragment of a class diagram, a sequence diagram, a program, a test case, or a project plan, for example, is a potential component. In order to maximise granularity of reuse, models are packaged together. For example, in RSE, there is a clear thread of development running from use cases down to coded programs, and RSE suggests that they can be packaged together on that basis. Reusing a use case means reusing all classes analysed in the analysis model (for the use case), the sequence diagram in the design model, the programs in the implementation model and the test cases in the test model.

One key issue in reuse is genericity. In order to make something reusable, it is important to make it generic; the more generic an artefact is the more reusable it becomes. Therefore, RSE reminds us of a range of 'variability mechanisms' that are at the disposal of developers. These include Inheritance for classes, Uses and Extends for use cases, Parameterisation for classes, Configuration and Module-interconnection languages and other CASE Tools related facilities. Using some of these mechanisms in all stages of development, RSE intends to make all artefacts more generic and reusable.

4.2.3.2 Identification and Validation of Components

Since RSE's definition of component is rather loose, no hard and fast rules for identification and verification of components are provided. However, from the SDP model of RSE, it can be ascertained that identification and validation of components will take place in Application Family Engineering and Application System Engineering sub-processes respectively.

4.2.4 Evaluation of the System Development Process of RSE

Table 4-5 summarises the evaluation of SDP of RSE.

Table 4-5 Evaluation of RSE System Development Process

The MAP Criteria	RSE
Feasibility Analysis	SDP of RSE explicitly requires the feasibility analysis to be carried out at the beginning of the development stages.
Business Modelling	RSE emphasises the importance of business analysis for software reuse, and it often refers to the object-oriented business process modelling approach discussed in (Jacobson et al, 1994).
Requirement Analysis	RSE mainly deploys the requirements analysis techniques mentioned in OOSE, such as use case modelling and domain object modelling. New concepts are also added, such as various reuse mechanisms (see Section 4.2.3.1).
System Analysis	System analysis in RSE revolves around robust analysis or class modelling. In OOSE, sequence diagrams (together with state diagrams) are used in the design stage. In RSE, sequence diagrams are produced in the analysis stage.
Logical Architecture	The robust analysis dictates the logical architecture in RSE. The separation of classes into different types, namely, boundary, control and entity objects, contribute to the creation of a layered architecture.
Physical Design	RSE provides few guidelines for developing a physical design of the system.
Component Search	RSE has an entire sub-process and indeed a business department devoted to developing and managing components.
Component Certification	This is part of the management of components.
Component Implementation	Discussions are general and largely restricted to how classes can be implemented in programming languages such as C++ and Smalltalk.
Application Assembly	Not provided.
System Testing	RSE refers to existing literature, for example, OOSE.
System Delivery	Not provided.

4.3 Evaluation SELECT Perspective (Perspective)

A summary of SELECT Perspective is provided in Section 3 of Appendix II.

4.3.1 Correlations Between the Three Elements of a Method

In terms of the three elements of a method suggested by the MAP framework, SELECT Perspective is a complete method. In addition, the following correspondence between these elements can be observed.

- There are clear mappings between the models and the system development process. SELECT Perspective clearly defines the stages of developments in terms of models produced in each stage, which are mostly in the solution process.
- Some mappings between the models and software architecture can also be observed; for example, models such as the deployment model are all about architectural modelling.
- The component process is largely about architectural analysis, and therefore, there is also a good level of interconnectedness between the system development process and the software architecture.

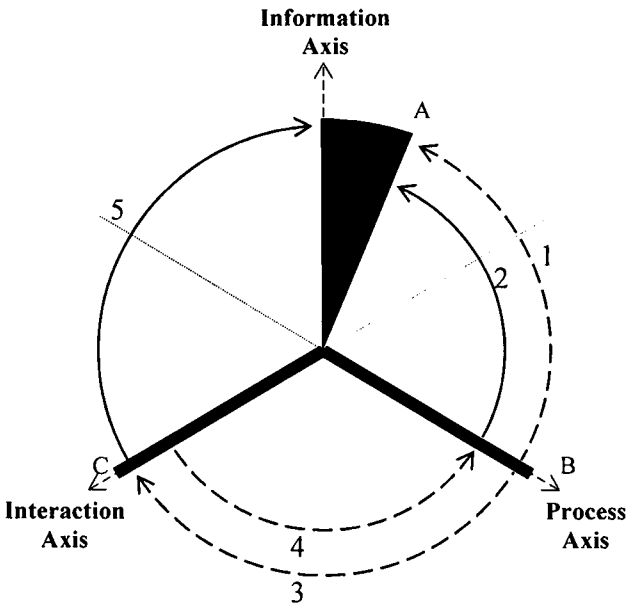
4.3.2 Evaluation of System Modelling

System Modelling of SELECT Perspective is described in Section 2.2 of Appendix II. The following is the IPI Matrix for the models in SELECT Perspective.

The following conclusions can be drawn about the coverage of models used by SELECT Perspective.

- SELECT Perspective provides a global process model (the use case model). Both LDS and class models can be regarded as information models (hence the triangular area), and although there is no explicit global model showing interaction between the system and its environment, the list of events can be considered as such.
- Out of six possible detailed contextual models, this method provides only two. As with UML, detailed contextual modelling in perspective is largely centred on use case realisation and state modelling.
- There are three abstract contextual models observed in this method.

Figure 4.2 IPI Matrix showing SELECT Perspective models



A. Logical Data Model and Class Diagram
B. Use Case Model
C. Events – although Perspective does not have a separate global model for this axis, an extensive use of events in the early stage of analysis and integration with other UML models in later stages justify inclusion of this model
1. Textual analysis of use case descriptions gives some indication of classes and objects that participate in the collaboration
2. Object Interaction Model uses UML Sequence and Collaboration diagrams
3. and 4. SELECT Perspective suggests that there is a general correspondence between a use case and an event, usually one-to-one. But the relationship is not explored in great detail.
5. State Model

4.3.2.1 Evaluation of Use Case Modelling

The modelling concepts of use case modelling in SELECT Perspective are similar to those used in RSE (see Section 4.3.2.1).

Table 4-6 Evaluation of SELECT Perspective Use Case Modelling

Criterion	Guidelines	Rigour
(a) Completeness of use cases	"Use cases are identified along chains of event-related activity" (p. 68). Alternatively, business processes in the business process model also provide the basis for identification of use cases.	2
(b) Completeness of actors	Some actors can be identified using Joint Application Development sessions. In addition, the statement of purpose of the system is examined to find out about	2

	responsibilities of those who use the system. Alternatively, business actors, identified in the business process modelling, also provide candidates for various actors (p. 67-68).	
(c) Completeness of associations	Associations are identified with use cases. Perspective categorises different types of actors, such as external and internal actors, which helps discover complex associations. However, it falls short of suggesting how completeness of associations can be achieved.	1
(d) Completeness use case relationships	"Alternative courses can be modelled as use case extensions. Extensions are commonly used to partition error and exception functionality" (p. 77)	1
(e) Minimality of use cases	Not explicit, though it can be inferred that since use cases tend to have clear correspondence with business processes, which are defined as atomic tasks performed by an actor at a place, it may help prevent duplicate use cases slipping into the model.	1
(f) Minimality of actors	See (d).	1
(g) Minimality of associations	See (c)	1
(h) Minimality of use case relationships	Perspective generally suggests not using too many use case relationships. This is not methodical; however it goes some way to eliminate unnecessary use case relationships.	1
(i) Correctness of use cases	Perspective emphasises the need for user participation in the development of the use case model, and it is supported by some prototyping.	1
(j) Correctness of actors	See (g).	1
(k) Correctness of associations	It is not clear how this correctness is ensured.	0
(l) Correctness of use case relationships	See (g).	1
(m) Non-redundancy of use cases	Not explicit, though it can be argued that the use of <<uses>> and <<extends>> relationships help factor out the common and unnecessary elements in the uses cases.	1
(n) Non-redundancy of actors	The use of generalisation/specialisation for the analysis of actor roles can help remove redundancy in actor roles.	2
(o) Non-redundancy of associations	No guidelines are provided.	0
(p) Non-redundancy of use case relationships	It is not clear how this is done in Perspective.	0

	Total	16
	Strength of the Perspective use case modelling technique	50

4.3.2.2 Evaluation of Class Modelling

Perspective has two models occupying the region around the information axis of the IPI Matrix, namely LDS and class model. Since LDS in Perspective is used to translate OO classes into Relational entities, the evaluation here will largely concentrate on the modelling technique for class modelling. The modelling concepts of class modelling in SELECT Perspective are similar to those used in RSE (see 4.2.2.2).

Table 4-7 Evaluation of SELECT Perspective Class Modelling

Criterion	Modelling Guidelines	Rigour
(a) Completeness of classes	Candidates for classes are first identified by noun analysis of the requirements statement and/or business services. For classes in real-time systems, events are examined. (p. 93) Classes are then analysed for their participation in use case realisations.	2
(b) Completeness of attributes	Attributes are identified mainly by analysing the business concepts in use case descriptions and requirements statements (p. 93). Classes generally have more than one attribute (p. 119).	1
(c) Completeness of operations	Class operations should reflect services. Again, use case realisation is crucial for completeness of class operations. Usually, classes do not have more than seven services (p. 119).	2
(d) Completeness of inheritance relationships	"Specialising involves examining a particular class for different ways in which its member objects can be split into subclasses. Generalisation involves searching for different classes that have some characteristics in common."	1
(e) Completeness of associations	"Associations identify good paths for communication between objects." Associations are often modelled as classes if they involve some attributes. "Aggregation relations are used to model whole-part relationships between objects" (p. 108). Classes may be either generalised to create superclasses or specialised to create subclasses.	2
(f) Minimality of classes	Classes should be checked for duplication and if necessary a generalised class should be used.	2
(g) Minimality of attributes	No guidelines are provided.	0
(h) Minimality of	It can be inferred that use case realisation will provide	1

operations	opportunities to do this.	
(i) Minimality of inheritance relationships	Classes can be specialised in many ways. "However, like most powerful concepts, inheritance should be used sparingly." Quoting Rumbaugh et al (1991), Perspectives suggests that "[a]n inheritance hierarchy that is two or three levels deep is certainly acceptable; ten levels deep is probably excessive; five or six levels, may or may not be proper."	2
(j) Minimality of associations	No guidelines are provided.	0
(k) Correctness of classes	JAD sessions and prototyping are recommended during this modelling, providing opportunities to validate various aspects of the model. Exactly how this should be done is not specified.	1
(l) Correctness of attributes	See (i).	1
(m) Correctness of operations	See (i).	1
(n) Correctness of inheritance relationships	No guidelines are specified.	0
(o) Correctness of associations	See (i).	1
(p) Non-redundancy of classes	Classes should be checked for duplication and if necessary a generalised class should be used.	2
(q) Non-redundancy of attributes	When classes are converted into relational entities in Logical Data Structure, Relational Data Analysis rules are applied. This analysis requires removing redundant attributes and relationships.	2
(r) Non-redundancy of operations	No guidelines are specified.	0
(s) Non-redundancy of inheritance relationships	No guidelines are specified.	0
(t) Non-redundancy of associations	See (n).	2
	Total	23
	Strength of the Perspective Class/LDS modelling technique	58

4.3.2.3 Evaluation of Event Modelling

The main concept used in this model is Event.

Figure 4.3 Evaluation of SELECT Perspective Event Modelling		
Criterion	**Modelling Guidelines**	**Rigour**
(a) Completeness of events	Perspective events are linked with Business Processes Modelling. Perspective suggests that events are signalled by the arrival of some data and time. Based on these events, Perspective identifies Elementary Business Processes (EBPs), a unit of work done by a single person at a time in a place. Since EBPs are later interfaced with use cases, there is an opportunity to crosscheck the events with use cases and classes.	2
(b) Minimality of events	See (a).	2
(c) Correctness of events	No guidelines are provided for validation the events by users.	0
(d) Non-redundancy of events	There are no clear guidelines on eliminating overlapping events.	0
	Total	4
	Strength of Perspective State Model	50

4.3.2.4 Evaluation of Object Interaction Modelling

The main concepts of Perspective Object Interaction Modelling (OIM) are similar to those used in RSE Interaction Diagram (4.2.2.3).

Table 4-8 Evaluation of SELECT Perspective Object Interaction Modelling		
Criterion	**Modelling Guidelines**	**Rigour**
(a) Completeness of participating objects	Use case descriptions are first analysed for the sequence of steps and participating objects. (p.133) Control objects are introduced to minimise coupling between business objects and interface objects. (p.137)	1
(b) Completeness of messages	Some of the steps of the use case description indicate messages. (p. 133)	1
(c) Completeness of flows	It is mainly derived for the use case description.	1
(d) Minimality of participating objects	Textual analysis of use case descriptions gives some indication of participating objects, and minimality of these objects can only be determined implicitly at best.	1
(e) Minimality of messages	No guidelines are provided	0
(f) Minimality of flows	Perspective discusses the problem of "fork and stair structures" in distributing control over objects, and suggests that, in general terms, the choice depends on the strength of relationships between participating objects.	1

	Total	5
	Strength of Perspective OIM modelling technique	42

4.3.2.5 Evaluation of State Modelling

The main concepts used here are also same as those used in RSE State Diagram (see Section 4.2.2.4)

Table 4-9 Evaluation of SELECT Perspective State Modelling

Criterion	Modelling Guidelines	Rigour
(a) Completeness of states	The concept of event is used in various stages of development such as business process modelling and use case modelling. Perhaps for that reason, Perspective does not state how they are identified for this modelling.	2
(b) Completeness of events/transitions	In the life-cycle approach, states can be identified analysing how attribute values and links change over a span of time. Alternatively, they can be identified from the messages objects receive from control objects in the sequence and collaboration diagrams.	2
(c) Minimality of states	No guidelines are provided for elimination of unnecessary states.	0
(d) Minimality of events/transitions	To some extent, it is implicit from (b).	1
	Total	5
	Strength of Perspective State Model	63

4.3.3 Evaluation of Architecture

The main features of the reference architecture recommended by SELECT Perspective are summarised in Section 3.2 of Appendix II.

4.3.3.1 Definition of Component

Perspective regards components as executable code units that provide services through their published interfaces. Since Perspective views a system as layers of services in architectural terms, the concept of service is crucial to its component modelling. 'Service' is defined as a group of related operations that provide useful functionality to consumers. Services are grouped into physical units called service packages. Components implement these services and groups of components that support a service package are called component packages.

The architectural model suggested by Perspective is neither radically new nor technically complex. It is a simple tiered architectural model, much like the classic MVC (Model-View-Control Model) of Smalltalk (Bennett et al, 2002). What is interesting, however, is the integration of the concept 'service' into this architectural model. The term 'service' is used in Perspective to mean a 'collection of related functionality' that can only be 'accessed through a consistent interface'. In OO terms, a service generally means a coherent group of class operations that is meaningful in business sense. Therefore, the granularity of a service is typically higher than normal class operations, coming somewhere closer to the granularity of a use case. Alternatively, services are similar to 'responsibility' of CRC approach (Bellin and Simone, 1997; Wirfs-Brock et al, 1990) in terms of granularity. In order to construct a higher level object that would provide higher-level operations, rather than low-level class operations, Perspective introduces control objects (Jacobson et al, 1992) that encapsulate groups of operations and act as a kind of interface for the services. Therefore, service classes, not ordinary classes, are the basic material of software architecture in Perspective. Control classes that provide at least one service are called service classes, and each service layer in the Perspective architectural model is made up of service classes.

The Perspective architectural model is a tried and tested model for middle-sized enterprise business applications.

4.3.3.2 Identification and Validation of Components

Perspective suggests that identification of business-oriented components draws from a number of sources including, domain knowledge, business process models, solution project feedback, generic models and patterns, legacy systems and models, legacy database and packages. The technique is more of a collection of buzzwords than an incisive modelling approach.

4.3.4 Evaluation of System Development Process

SELECT Perspective has a reasonably complete development process, from feasibility and business analysis to implementation and testing. It also emphasises user involvement at various stages of development. The development stages are clearly defined in terms of input and output processes.

Table 4-10 Evaluation of SELECT Perspective System Development Process

The MAP Criteria	SELECT Perspective
Feasibility Analysis	At this stage, the scope of development is defined in terms of the system it proposes, business justification and possibility of reusing existing components.
Business Modelling	The business process models of the current and the new system are produced.
Requirement Analysis	This is done mainly through business process, use case and class models.
System Analysis	Class modelling, object interaction modelling and state modelling are carried out.
Logical Architecture	A component diagram showing service components of the three types, namely, user services, business services and data services is produced.
Physical Design	The component diagram is revised by taking into accounts various considerations such as implementation environment, legacy systems etc. A deployment diagram is also produced.
Component Search	There is an explicit process of component search.
Component Certification	Not provided.
Component Implementation	Discussions are restricted to reuse of legacy systems.
Application Assembly	Same as Component Implementation above.
System Testing	Only general discussions are provided.
System Delivery	Same as System Testing above.

4.4 Evaluation of Catalysis

A summary of Catalysis is provided in Section 4 of Appendix II.

4.4.1 Correlations Between the Three Elements of a Method

Of the three elements of a method, only system modelling is described in detail; discussions on SDP and system architecture are mainly about process and architectural patterns. As far as the correlations between the three elements are concerned, only general mappings can be ascertained from the SDP of Catalysis, as indicated in Figure 4-7 of Appendix II.

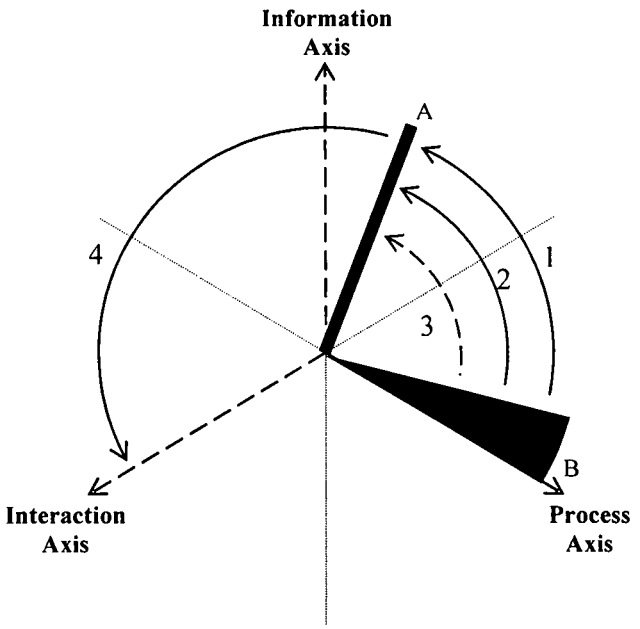
- Since the Catalysis development stages are rather generic, the mappings between the models and the development stages are not detailed
- Catalysis does not suggest a reference architecture, although some models clearly deal with architectural issues
- Catalysis does have development stages that focuses on architectural issues

4.4.2 Evaluation of System Modelling

Figure 4.4 shows the IPI Matrix for Catalysis models, from which the following conclusions can be drawn:

- Catalysis has only two global models; although State Charts refer to events, there is no global model in Catalysis that focuses on interactions between the user and the system at the global level.

Figure 4.4 IPI Matrix showing analysis models of Catalysis



A. Static Model: shows classes/types, their attributes and operations
B. Behavioural Model: shows the actions as well as types
1. Interaction Model: shows collaboration between objects
2. Snapshot: shows state of objects, links and attribute values at a time
3. Types in the behavioural model are same as types in the static model
4. State Model: shows time-oriented behaviour of an object/system

- There only one abstract contextual model in Catalysis, indicating that there is very little common elements between global models
- There are three main detailed contextual models in Catalysis, two of which occupy the same region between the Information and Process

axes on the IPI matrix. State Charts refer to events, and because there is no global interaction model in Catalysis, the arrow for State Chart in the IPI Matrix points to the Interaction axis.

4.4.2.1 Evaluation of Behavioural Models

Behaviour of a component or a system is described "by specifying the component's type: a list of actions it can take part in and the way it responds to them." Again, OCL expressions are used to capture the effects of actions. Importantly, Catalysis does not allocate operations to classes or types straight away. It keeps the type and actions separate in the behavioural model, although they make implicit references to each other (Chapter 3, e.g. p. 139). The main concepts are: Action, Type, Action Refinement and Type Refinement.

Table 4-11 Evaluation of Catalysis Behavioural Model

Criterion	Modelling Guidelines	Rigour
(a) Completeness of actions	Catalysis implies the use of grammatical analysis of a problem description, where verbs are candidates for actions. Pre- and post-conditions of an action are depicted using a pair of snapshots.	2
(b) Completeness of types	See (c) in Table 4-12.	1
(c) Completeness of action refinements	Refinement of action and types are interdependent. If a collaboration for an action involves more than one type, usually the action is further decomposed, until no further refinement of types is possible and actions become messages between classes. In Catalysis all actions and types are refined in that fashion.	2
(d) Completeness of type refinements	See (c)	2
(e) Minimality of actions	Verbs that are not mentioned in the specification text are not candidates for actions.	1
(f) Minimality of types	See (h) in Table 4-12.	1
(g) Minimality of type refinements	Implicit from (c); however there are no specific checks to ensure this minimality.	1
(h) Minimality of action refinements	Implicit from (c).	1

(i) Correctness of types	See (m) in Table 4-12.	1
(j) Correctness of actions	Since there is little emphasis on the user participation through prototyping or a kind of user interface modelling, users may not be albe to confirm whether an analysis of actions is correct.	0
(k) Correctness of type refinements	See (i).	0
(l) Correctness of action refinements	See (i).	0
(m) Non-redundancy of types	See (r) Table 4-12.	1
(n) Non-redundancy of actions	Catalysis discusses various factoring techniques for reducing redundancy in action specifications (p. 117-126). For example, it suggests that some pre- and post-conditions can be made into an attribute with a simple invariant; and specifications of common pre-conditions can be shared, and so on.	2
(o) Non-redundancy of type refinements	Although Catalysis discusses extensively about how to use refinement of types and actions, it does not suggests clearly when the use of refinement will become unnecessary.	0
(p) Non-redundancy of action refinements	See (n)	0
	Total	15
	Strength of the Catalysis behavioural modelling technique	47

4.4.2.2 Evaluation of Static Model

Catalysis suggests models have static, dynamic and interactive parts. The static model aims "to provide a vocabulary in which to describe actions, which include interactions in a business, between users and software, or between objects in side the software" (p. 45, 77). The main concepts used in the diagram are Objects, Attributes, Types, Associations and Invariants.

Table 4-12 Evaluation of Catalysis Static Model

Criterion	Modelling Guidelines	Rigour
(a) Completeness of objects	"Anything that can be identified as an individual thing, physical or conceptual, can be modelled as an object; if you can count it, distinguish it from another, or tell when it is created, it is an object" (p. 49). It is not clear how this completeness is ensured.	1

(b) Completeness of attributes	"The state of an object, the information that is encapsulated in it, is modelled by choosing suitable *attributes*. In constructing a model, we choose all the attributes that we need to say everything we need to say about the object." Snapshots are used to envisage how values of some attributes change at various points in time; for example, before and after an action has taken place, changing the state(s) of some object(s). (p. 50, 80)	2
(c) Completeness of types	Objects, their links and attribute values in the snapshots, are generalized into types, associations and attributes of the types (p. 57).	2
(d) Completeness of associations	"A pair of attributes that are inverses of each other, usually drawn as a line connecting two types." This means that two objects holding attribute values referring to each other imply an association between the type(s) of the objects. (p. 61)	2
(e) Completeness of invariants	Catalysis makes extensive use of invariants, "a Boolean (true/false) expression that must be true for every permitted snapshot." (p. 67) For example, an instructor who does not have the right qualification cannot be allocated to teach a course. Catalysis use Object Constraint Language to express such invariants.	2
(f) Minimality of objects	Not all objects identified are interesting, and "the behaviours that we wish to describe determine which objects and properties are relevant." This does not however suggest how the behavioural model itself can be validated. In the book, Catalysis uses a textual description of a case study as a basis for this exercise. (p. 49)	1
(g) Minimality of attributes	Catalysis suggests that exact formulation of attributes is unnecessary as long as there are operations that can provide the required information. It also emphasises the use of parameterised attributes which it says eliminate the need to normalise attributes and make the model simpler and more natural.	1
(h) Minimality of types	Catalysis discusses various ways to join types, i.e. combine specifications of types if they are thought to be describing the same thing from different angles. It is not explicitly concerned with removing unnecessary types.	1
(i) Minimality of associations	Implicit from (d).	1
(j) Minimality of invariants	Catalysis talks about combining invariants of classes, where the purpose is to simplify, rather than optimise the specification. Since it does openly discuss the issue, some grade can be awarded.	1
(k) Correctness of objects	Catalysis put little emphasis on user involvement in its modelling activity. In its case study, it uses a	0

	small interview transcript and, elsewhere a textual description, as a starting point for the analysis. It is difficult to see how Catalysis models are approved by users.	
(l) Correctness of attributes	See (k)	0
(m) Correctness of types	See (k)	0
(n) Correctness of associations	See (k)	0
(o) Correctness of invariants	See (k)	0
(p) Non-redundancy of objects	See (k)	0
(q) Non-redundancy of attributes	See (g)	1
(r) Non-redundancy of types	See (h)	1
(s) Non-redundancy of associations	No guidelines are provided.	0
(t) Non-redundancy of invariants	In some cases, Catalysis argues, deliberate redundancy of specifications can help simplify the models.	1
	Total	17
	Strength of the Catalysis static modelling technique	43

4.4.2.3 Evaluation of Interaction Models

These models describe "the most interesting aspect of any design [that] lies in the interactions among the objects: the way that the net behaviour resulting from their collaborations realises some higher-level function when they are configured together in a particular way." (p. 153) Collaborations can be a refinement of an abstract action or "a design to maintain invariants between objects" (p. 172). The main modelling concepts are: Collaboration, Message and Flow.

Table 4-13 Evaluation of Catalysis Interaction Models

Criterion	Modelling Guidelines	Rigour
(a) Completeness of collaborations	Candidates for type collaborations are actions identified previously in static and behavioural models. Here responsibilities are assigned to classes by hierarchically decomposing actions and allocating smaller-grained operations to types or classes. Constraints involving multiple types are enforced. Completeness of collaborations depends upon how types and actions are decomposed.	1

(b) Completeness of messages	'Scenarios' of an action are described. Then by following the steps in the descriptions, messages between collaborating objects are identified. (p. 177)	1
(c) Completeness of flows	See (b).	1
(d) Correctness of collaborations	No guidelines are provided.	0
(e) Correctness of messages	Implicit from (b).	1
(f) Correctness of flows	Implicit from (b).	1
	Total	5
	Strength of the Catalysis interaction modelling technique	42

4.4.2.4 Evaluation of Snapshot

The main concepts used in this modelling are: participating Objects and Messages passed between objects.

Table 4-14 Evaluation of Catalysis Snapshot

Criterion	Modelling Guidelines	Rigour
(a) Completeness of participating objects	Usually these are identified from requirements specification, user interview transcripts etc. Nouns in the descriptions are candidates for objects.	1
(b) Completeness of messages	Messages are identified from the verb analysis.	1
(c) Correctness of participating objects	Not provided.	0
(d) Correctness of messages	Not provided.	0
	Total	2
	Strength of the Catalysis snapshot modelling technique	25

4.4.2.5 Evaluation of State Charts

The main concepts used in Catalysis State Charts are: States and Transitions/Events.

Table 4-15 Evaluation of Catalysis State Charts

Criterion	Modelling Guidelines	Rigour
(a) Completeness of states	States are generally identified from attributes and invariants. Catalysis does not say how this is done. "Sometimes it is easy to see distinct states that an object progresses through over its	1

	lifetime." (p. 126)	
(b) Completeness of events/transitions	Actions provide the basis for identification of transitions. (p. 127)	1
(c) Correctness of states	No guidelines are provided.	0
(d) Correctness of events/transitions	No guidelines are provided.	0
	Total	2
	Strength of the Catalysis state modelling technique	25

4.4.3 Evaluation of Architecture

Instead of proposing a single main reference architectural model, Catalysis provides general discussions covering issues such as architectural views, physical and logical architecture system qualities affected by architectural design decisions, architectural styles, and patterns such as four-tier business architecture.

4.4.3.1 Definition of Component

Components can be either executable or non-executable. Executable components can be implemented using popular OO programming languages such as Java. Non-executable components include a range of development artefacts, in particular, design pattern and frameworks (Gamma et al, 1995).

4.4.3.2 Identification and Verification of Components

Components are identified by recursively decomposing the system. In Catalysis, each class, or type, or collaboration of classes to fulfil an action is potentially a component. In this sense, identification and validation of components are done rather informally.

4.4.4 Evaluation of System Development Process

Catalysis does not provide a detailed SDP model; instead it suggests a general outline of development process and for specific projects, it recommends the use of process patterns. The main stages of development are: Requirements Specification, System Specification, Architectural Design and Component Internal Design.

4.5 Evaluation of KobrA

A summary of KobrA is provided in Section 5 of Appendix II.

4.5.1 Correlations Between the Three Elements of a Method

KobrA mainly focuses on system modelling. Its treatment of the development process is fairly minimal, while its discussions on software architecture primarily deal with component design. KobrA does not provide any reference architecture, leaving it to analysts to choose an appropriate architectural pattern (Kircher and Jain, 2004).

- KobrA is similar to RSE in the sense that both regard the development process as a series of development of models. Therefore, the system development process and the model development process are very much intertwined. At any rate, the development process described by KobrA, the product line engineering process, is limited.
- There is general correspondence between system modelling and software architecture: the structural model explicitly deal with architectural design issues, although KobrA does not provide any reference architecture.
- The correlations between architecture and the development process are made clear through the modelling activities; the structural model and the containment tree deal with the architectural aspects of the system.

4.5.2 Evaluation of System Modelling

Figure 4.5 shows the IPI matrix for the models used in this method.

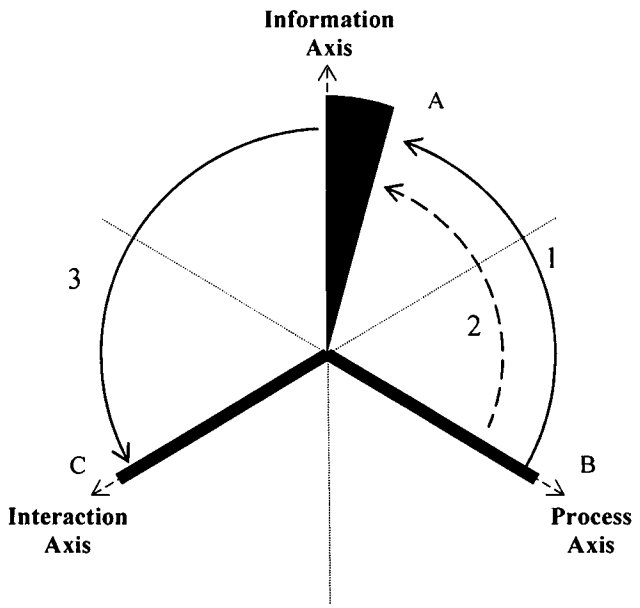
As far as the modelling coverage is concerned, the following conclusions can be drawn:

- KobrA provides three global models dealing with information, interactions and processes of systems. The suggestion by KobrA that statecharts can be used to model interactions between the user and the system at the global level, traced back to OO methods such as Fusion (Coleman et al, 1993), is questionable. Two points can be raised against this proposition. Firstly, Fusion and KobrA use conveniently simplified ATM and library systems to show that the global behaviour of the system can be captured in terms of state-dependencies. An experiment in this research shows that when the behaviour of the system is richer, the diagram either becomes so complex

that it hardly makes sense or the behaviour cannot be adequately captured (Section 16.3). For this reason, the MAP framework suggests that time-dependent analysis of the system is better suited for contextual models only (Section 3.4.2.2). Secondly, statechart is not a model that aims to show input/output-based interactions between the system and the user. However, since it contains all major events of the system, and any detailed description of these involves inputs and outputs, it qualifies as a partial interaction model.

- There is only one significant abstract contextual model in this method.
- Out of six possible detailed contextual models, there are only two main models observable in this method.

Figure 4.5 IPI Matrix Showing Main KobrA Models



A. Structural Model: shows Class/Komponent diagram and Object diagrams, where 'classes' are usually operationless, while Komponents are not.
B. Functional Model: shows textual specification of effects of Komponent operations
C. Enterprise Model (Interaction Diagram): KobrA suggests using either an activity diagram, use case diagram or interaction diagram for enterprise modelling, where only the interaction diagram can be counted as an Interaction model.
1. Interaction Model (Collaboration Diagram)
2. Activities in the process model referring to Komponents in the structural model
3. Behavioural Model: Statechart diagram / table

The fact that there are very few contextual models is an indication that there is very little cohesion between KobrA models, and also that they are fragmented. Perhaps to address this, KobrA discusses one unique feature of modelling: the so-called Intra-

diagram and Inter-diagram rules, which specify the mapping of elements between models. For example, some intra-diagram rules for the class diagram require that only Komponents (not ordinary classes) should have operations, several Komponents must show all or none of their operations, and the client Komponents need to show the operations the server invokes. Some inter-diagram rules between the structural and behavioural models include attributes of classes in both models being consistently specified and the states being expressible in terms of classes (p. 430). These rules and meta-models are useful for creating CASE Tools and ensuring consistency between models. This is to KobrA's credit. However, these static rules are no replacement for modelling dynamics of the system; only the appropriate contextual models serve this purpose.

4.5.2.1 Evaluation of Functional Model

The functional model of KobrA is a set of specifications of operations, as in Fusion (Coleman el al, 1993). This modelling is often preceded by an enterprise model that involves activity/ use case/ interaction modelling (p. 157, 169-170). The main concept is Operation. Since any reasonable specification of an operation is accompanied by two clauses, Assume and Result, identification of clauses is also considered in this evaluation.

Table 4-16 Evaluation of KobrA Functional Model

Criterion	Modelling Guidelines	Rigour
(a) Completeness of operations	Operations are identified by examining the messages sent to instances of the Komponent (p. 116).	1
(b) Completeness of assumes	Parameter types and return types of operations are identified. Intended effects of operations are identified in object, class, statechart diagrams and parameters. A snapshot object diagram can be used to model the configurations objects before and after the operation. "Identify all appropriate assumptions." (p.116)	1
(c) Completeness of results	"Formulate result and assumes clauses." For all initial assumptions in operation specifications there are final values that satisfy the result clause. (p116)	2
(d) Minimality of operations	Summarise operations of server Komponents. (p. 116)	1
(e) Minimality of assumes	Parameter types and return types are summarised.	1
(f) Minimality of results	See (e).	1
(g) Correctness of	No guidelines are provided.	0

operations		
(h) Correctness of assumes	No guidelines are provided.	0
(i) Correctness of results	No guidelines are provided.	0
(j) Non-redundancy of operations	No guidelines are provided.	0
(k) Non-redundancy of assumes	No guidelines are provided.	0
(l) Non-redundancy of results	No guidelines are provided.	0
	Total	7
	Strength of the KobrA functional modelling technique	29

4.5.2.2 Evaluation of Structural Model

KobrA makes use of a few diagrams for this model, class diagram, object diagram and containment hierarchy. The main concepts used in this model are Class/Komponent, Attributes, Operations, Association and Containment Hierarchy. The following table shows the criteria applicable to this model, and a summary of evaluation of the modelling technique used in this modelling.

Table 4-17 Evaluation of KobrA Structural Model

Criterion	Guidelines	Rigour
(a) Completeness of classes / Komponent	"Based on the functionality and behaviour, add data types for any entities needed to store persistent state or data. Indicators of the need for such data types are states in the behavioural model or post-conditions in the functional model (p.133)" "All roles interacting with the system to be built as well as all entities involved in this interaction (p. 169)" are candidates for classes. Two criteria are applied to distinguishing simple classes from Komponents: multiplicity and granularity. Komponents tend to have fewer instances and higher granularity. (p. 151-152)	1
(b) Completeness of attributes	"Add appropriate logical attributes to all classes. Add any missing logical attributes required in any of the operation specifications or any of the statechart states" (p. 115). "Get and Set operations pairs can be modelled as logical attributes" (p.134).	1
(c) Completeness of operations	"Every message sent to an object in an interaction diagram must correspond to an operation of the corresponding class in the class diagram" (p. 134) During activity modelling of the system operations are identified. These are added as operations to the system Komponent (p. 169 and 135).	2
(d) Completeness of associations	"Add essential associations between all classes. Add any missing associations needed in any of the operation specifications to navigate between instances." (p.115)	2

(e) Completeness of containment hierarchies	KobrA only suggests that Komponents are identified at the end of realisation modelling, by looking at multiplicity and granularity of the classes. The only way to partially ensure that all Komponents and classes in the hierarchy have been completely identified is through interaction modelling. (p. 151-152)	1
(f) Minimality of classes	Remove any model elements not needed in the functional model or behavioural model. (p. 115)	2
(g) Minimality of attributes	See (f).	2
(h) Minimality of operations	See (f).	2
(i) Minimality of associations	See (f).	2
(j) Minimality of containment hierarchies	Simple data structures should not be treated as Komponents (p. 151-152). KobrA discusses a technique on how to refactor containment hierarchy by adjusting it so that Komponents and classes have proper visibilities. (p. 153-154) However, there is no mechanism to ensure that the hierarchy of components and classes do not contain any unnecessary elements and levels of hierarchy.	0
(k) Correctness of classes/ Komponents	KobrA emphasises the role of User Interface artefacts in identifying classes, but it does not go as far as saying that the users need to participate to validate models being produced. Perhaps it is due to the nature of Product Line Engineering, which requires the analysts to have specialist domain knowledge of the application, and the analysis process is driven by generic requirements, rather than situation-specific requirements. Hands-on participation of users in the development of a framework, as in RAD, may largely be unnecessary. Still, KobrA does not say how the knowledge of the domain can be validated. Komponents have no 'real world' counterparts and it is up to analysts to decide whether an invented Komponent is correct; KobrA provides the two criteria, multiplicity and granularity, to differentiate classes from Komponents.	1
(l) Correctness of attributes	See (k).	0
(m) Correctness of operations	See (k).	0
(n) Correctness of associations	See (k).	0
(o) Correctness of containment hierarchies	The visibility rules of a Komponent tree provide the basis for determining the locations of Komponents within it. A containment hierarchy that is consistent with the visibility rules may not necessarily satisfy users' requirements.	1
(p) Non-redundancy of classes/	System analysis in KobrA is very much influenced by implementation-oriented design considerations; for example,	1

Komponents	refactoring of classes is very much a design activity which in KobrA is done during what can be called the system analysis stage. Addition and removal of classes or components are often done in the light of implementation. Therefore, there is no logical model that is robust and free from influences of implementation.	
(q) Non-redundancy of attributes	See (q).	1
(r) Non-redundancy of operations	See (q).	1
(s) Non-redundancy of associations	See (q).	1
(t) Non-redundancy of containment hierarchies	Not provided.	0
	Total	21
	Strength of the KobrA structural modelling technique	53

4.5.2.3 Evaluation of Activity Model

Activity diagrams are used to describe the algorithms of Komponent operations. (p. 123). The main concepts are: Activity/Subactivity and Flow.

Table 4-18 Evaluation of KobrA Activity Model

Criterion	Modelling Guidelines	Rigour
(a) Completeness of activities/subactivities	"For each operation, identify the subactivities needed to fulfill the operations specification effects. Classic structured decomposition techniques can be applied at this stage to determine the subactivities." (p. 135) Activities are then crosschecked with Komponent operations.	2
(b) Completeness of flows	First inputs to, and outputs from, each subactivity are determined. Then activities that provide and require input and output data from the activities are connected accordingly to create the flow.	2
(c) Minimality of activities/subactivities	By allocating subactivities to data type, unnecessary activities can be removed (p. 135).	2
(d) Minimality of flows	Implicit from (b).	1
(e) Correctness of activities/subactivities	No guidelines are provided.	0
(f) Correctness of flows	No guidelines are provided.	0

(g) Non-redundancy of activities/subactivities	No guidelines are provided.	0
(h) Non-redundancy of flows	No guidelines are provided.	0
	Total	7
	Strength of the KobrA activity modelling technique	44

4.5.2.4 Evaluation of Interaction Model

KobrA's interaction model also describes the algorithm by which an operation is realised, but from the objects' perspective, rather than the flows' perspective. Following the trend of Fusion, it suggests using UML collaboration diagram for this purpose. The main modelling concepts are: participating Objects, Messages and Flows.

Table 4-19 Evaluation of Interaction Model

Criterion	Modelling Guidelines	Rigour
(a) Completeness of participating objects	"Identify collaborations: Once tentative allocations have been determined, identify potential collaborations between objects for realising the unresolved activities (i.e. those currently at the leaves of the activity hierarchy). Consider modifying the data types, activities and allocations." (p. 135)	2
(b) Completeness of messages	"All the messages within collaborations represent operations of objects." (p. 136) Operations are then collated and specified.	1
(c) Completeness of flows	See (b) in Table 4-18.	2
(d) Minimality of participating objects	Objects must be instances of classes.	1
(e) Minimality of messages	Messages must correspond with the operations of classes.	1
(f) Minimality of flows	See (d) in Table 4-18.	1
	Total	8
	Strength of the KobrA interaction modelling technique	67

4.5.2.5 Evaluation of Behavioural Model

KobrA suggests using statechart diagram(s) and/or statechart table(s). The main modelling concepts are: State and Events/Transition.

Table 4-20 Evaluation of KobrA Behavioural Model		
Criterion	**Modelling Guidelines**	**Rigour**
Completeness of states	"Identify all externally visible, logical states of the Komponent. A visible state characterises the operations that can be performed on the Komponent under particular circumstances." (p. 117)	1
Completeness of events/transitions	State attributes of classes and Komponents are identified. Operations that can be executed in each state, and the invocation of the transition are identified. (p. 117)	1
Minimality of states	No guidelines are provided.	0
Minimality of events/transitions	No guidelines are provided.	0
	Total	2
	Strength of the KobrA statechart modelling	25

4.5.3 Evaluation of Architecture

Since KobrA does not give any reference architecture, nor discuss how a known architecture should be selected, it is left to the analysts to determine what is appropriate for a given application.

4.5.3.1 Definition of Component

KobrA components, or Komponents, have three parts: a specification, realisation and one or more implementation. KobrA views a system as a hierarchy of Komponents, which is expressed through a containment hierarchy. Since KobrA is based on the concept of product line engineering, the aim of reuse is not of individual components, but an entire application. Therefore, generic applications, called frameworks, are built with all possible variant features, and the application development in the traditional sense is all about choosing the right set of variant options and instantiating the application. KobrA, therefore, aims for total reuse.

4.5.3.2 Identification and Validation of Components

Domain modelling with an emphasis on identifying common and application-specific features is crucial for the development of frameworks. KobrA also discusses in great detail about how component hierarchy should be composed and rules applying the composition.

4.5.4 Evaluation of System Development Process

The system development process of KobrA is markedly different from all other processes investigated in this research because it is uniquely based upon the concept of product line engineering. Its development process is only described in terms of context realisation of the system followed by a series of specification and realisation in an iterative manner. Seen through the prism of the MAP evaluation framework, the development stages of KobrA are not clearly demarcated, and as a result, there are ambiguities about what analyst should be focusing on in each stage. KobrA also does not emphasise the user participation in the development process, which is also evident in modelling. The reuse is perhaps most effective with the product line approach because it is entirely geared towards full reuse of applications.

4.6 Experiment on the framework

When we attempted to publish a paper on the evaluation of UML using the MAP framework, a referee gave some comments relating to the repeatability of the evaluation exercise, i.e. if someone else has carried out the evaluation with the MAP framework, would they still get the same results? The paper was accepted for publication (Bielkowicz and Tun, 2003). Nevertheless, we decided to set up a small experiment to investigate the issue. The following sections discuss the experiment methods and results for the data gathered in the academic year 2003/2004. The experiment is repeated this year and the results are due in early June, 2005.

4.6.1 Objective of the Experiment

The main objective of the experiment is to find out whether the MAP framework, in particular the IPI matrix, can be applied independently by others who would then come to a similar conclusion by producing comparable IPI matrixes. For example, evaluation of the two popular methods, SSADM and UML, using the IPI matrixes in this research (see Figure 3.9 and Bielkowicz and Tun, 2003) suggest that SSADM has better coverage and more contextual models, both abstract and detailed.

4.6.2 The Experiment Method

Requirements

Participants of the experiment need to have working knowledge of the following:

- UML, in particular its models such as Use Case Model, Class Model, Use Case Realisation Model (Sequence Diagram and Collaboration Diagram) etc, and how the models are related to each other (OMG, 2003)
- SSADM, in particular its models such as Context Diagram, Data Flow Diagram, Entity Relationship Diagram, Entity Event/Access Matrix, Entity Life History, Effect Correspondence Diagram, and relationships among these models (Goodland and Slater, 1995; Bentley 1997; Weaver et al 1998)
- Participants also need to develop a good understanding of the framework, and possess the ability to think critically and conceptually.

Participants

In the experiment that took place at London Metropolitan University in the second semester of the academic year 2003-2004, the participants were students of the QC309 Advanced Systems Analysis and Design unit. All students had done various units on UML and SSADM in previous years of their studies. The 29 students taking part were asked to divide themselves into groups of between two and four students. This resulted in nine groups of participants.

Plan for the experiment

- Participants were first given a formal lecture and a tutorial reviewing various models of UML and SSADM by the author (in the first week of the second semester, 2003-3004)
- Participants were given a lecture on NIMSAD (Week 2)
- Participants were given a lecture on the MAP framework, with particular emphasis on evaluation of models using IPI Matrix (Week 3). The lecture notes included an extract from the published paper (Bielkowicz and Tun, 2002)
- Students assignments were issued, which required participants to produce the IPI matrixes showing appropriate UML and SSADM models and commenting on the coverage of and facility for checking consistency between the models in each method. The evaluation was a compulsory part of the assignment (it carried 25% of the overall mark of the assignment), countering the likelihood of participants not being motivated to take part.
- Participants handed in their work (Week 10) including the evaluation.
- Participants were given overall grades and comments on their work

4.6.3 Experiment Results

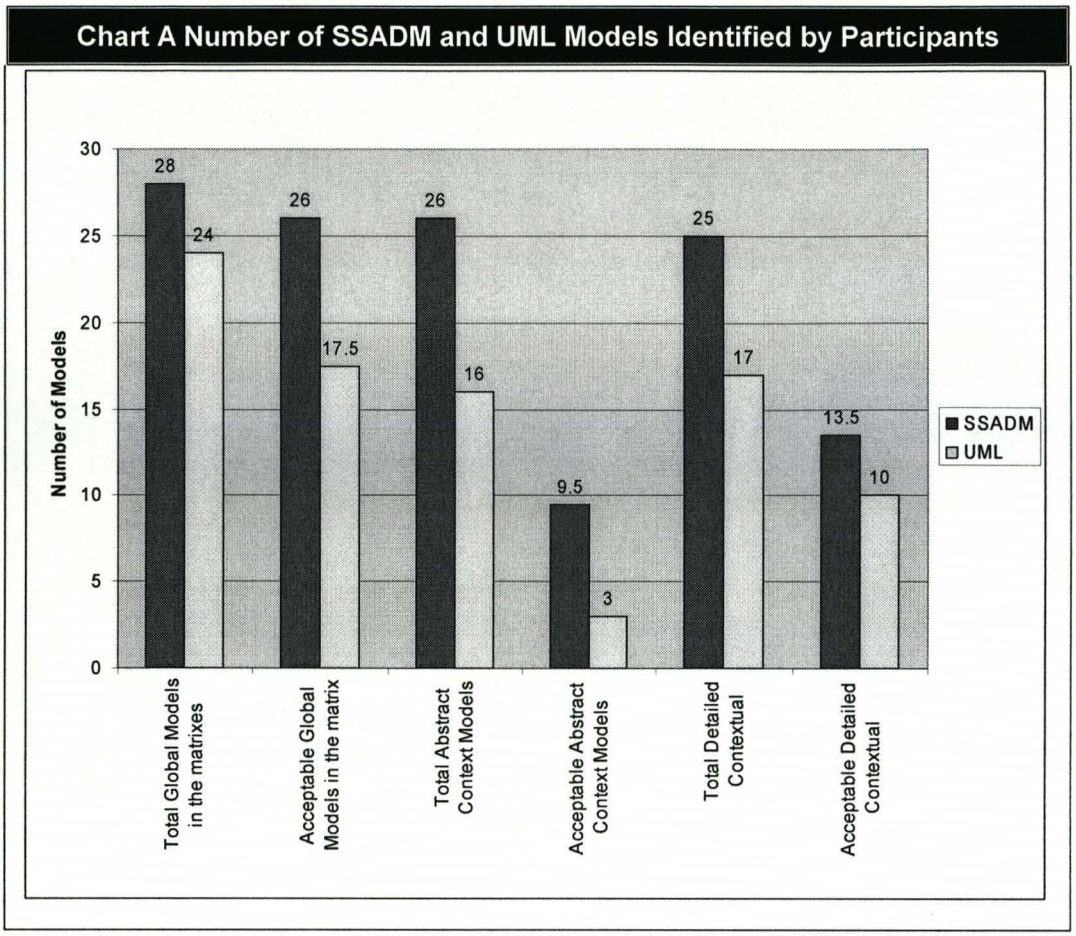


Chart A[5] shows total numbers of global, abstract contextual and detailed contextual models in SSADM and UML identified by the participants, together with the numbers of acceptable models. 'Acceptability' means that the categorisation of a model by participants is broadly in line with what the MAP framework suggests. For example, regarding activity diagram as a functional global model is acceptable, although use case modelling is usually the preferred choice for global system functionality modelling in UML, while assuming Entity Access Matrix as a global interaction modelling is not acceptable because it does not deal with the system's interaction with its users. When assessing students' work, there were a few cases in which wrong direction of arrows denoting contextual diagrams, confusion in denotation, and drawing made it difficult to ascertain exactly what was meant. If

[5] Raw data is attached in Appendix VI.

there was a good basis to believe that the entry was broadly correct, it was counted as 0.5, instead of 1. Otherwise, it was regarded as wrong.

The nine groups of participants should have identified at least 27 global models for a method (three global models per group per method). Assuming that there were three global models in each method, there should have been a maximum of 54 abstract contextual, and 54 detailed contextual models in each method. Participants found 28, one more than expected, global models in SSADM, although only 24 can be counted as acceptable. In that regard, UML did not fare well as only 17.5 global models were identified. Some participants noted that UML did not have a global interaction model, whilst some thought that either use case or activity might be counted as such. Relatively high numbers of global models can be explained by the fact that the lecture on the MAP framework included examples referring to some global and contextual models from both SSADM and UML[6], and that the global models are usually regarded as the most important by authors and teachers of methods.

Participants tended to be less confident about plotting contextual models onto the matrixes. They identified consistently more contextual models, both abstract and detailed, in SSADM than UML. The error rate here was rather high. Out of 26 and 16 attempts at abstract contextual models of SSADM and UML respectively, on average, only 36% and 19% were correct. For detailed contextual models, the figures went up to about 55% each. The fact the detailed contextual models are more 'visible' to analysts may explain this disparity.

It is clear from this chart that participants generally came to the conclusion that SSADM has more global models, abstract contextual and detailed contextual models than UML. In addition, participants were able to correctly map SSADM models, contextual models in particular, to the IPI matrixes more frequently than UML. Not only did participants find fewer relevant contextual models in UML, the number of times where no contextual model was identified for a contextual region (the region

[6] Since most participants did not know another major method well enough, it was not possible to explain the IPI matrix in great detail without making references to SSADM or UML. In order not to prejudice their judgements, some global and contextual models from both methods were mentioned as examples in the lecture.

between two axes) was 41, whilst for SSADM it was 25. These points were reflected in participants' commentaries. Four teams stated clearly that SSADM has a better coverage, as opposed to three teams for UML. The remaining two teams were undecided, although one of these teams was marginally in favour of SSADM. On the question of inter-model checks, five teams were largely in favour of SSADM, but only two of these teams thought that perhaps UML was in some way better.

Limitations and conclusions of the experiment

Participants had had exposure to both SSADM and UML, although they were hardly connoisseurs of system development methods. Errors and omissions in the matrixes can partly be accounted for by this point. Also, the sample size of the experiment was small; hence the experiment is repeated this year. If the general tendency, often instilled into students by teachers, authors and the general hype surrounding OO, to dismiss SSAMD as old and outdated, whilst promoting UML as more advanced, is taken into consideration, these findings are very revealing. The conclusions students reached are not through general hunches, but through careful analysis and critically thinking. It is therefore clear that the model evaluation method suggested by the MAP framework is to some extent repeatable, encouraging method users to think critically about methods.

4.7 Summary and Conclusion

Based on the evaluation of the existing CBSD methods presented in this chapter, the following conclusions can be drawn:

- RSE and Perspective not only provide all three elements of a method, but also offer good correlations between these elements. Coverage of SDP in both Catalysis and KobrA is relatively limited, and neither Catalysis nor KobrA provides a reference architecture. In this respect, RSE and Perspective are complete methods.
- IPI Matrixes of these methods show that Perspective models have the best coverage of all methods (Figure 4.2). There are three global models as well as some contextual models, which nevertheless fall far short of forming four complete circles around the axes. RSE (Figure 4.1) and Catalysis (Figure 4.4) both lack a global interaction model. Catalysis contextual models focus very much on the region between

the Information and Process axes, whilst RSE models have a fairer distribution across the IPI Matrix. KobrA (Figure 4.5) has three global models, but few contextual models. It is clear from these IPI Matrixes that coverage of models used by existing CBSD methods is generally poor. Models in these methods pay little or no attention to the analysis of user-system interactions, and these methods consistently fail to provide a sufficient number of abstract and detailed contextual models to help ensure internal and external consistencies of the global models

- Average strength of modelling techniques in RSE, Perspective, Catalysis and KobrA are 47, 53, 40 and 43 respectively. There seems to be a problem here: the more recent methods, and the ones that use more formal specification techniques, have less rigour in their modelling techniques. This paradox can be explained by two factors. The first factor is user participation: involving users in the development of models, through interface modelling, prototyping and so on, helps ensure internal and external consistency of system models. Since Catalysis and KobrA do not emphasise this, modelling techniques of these methods score consistently low for the semantic level criteria. The second factor is the failure to distinguish between expressiveness of a modelling language and guidelines for applying the language. Catalysis, for example, uses a barrage of modelling concepts: actions, action types, joint actions, localised actions, concurrent actions, external actions, internal actions, joint services, use cases, interactions, operations and message flows. These concepts all mean more or less the same thing, but Catalysis provides few guidelines on how to identify and validate them. This is like having a very comprehensive dictionary, but not knowing enough grammar to put words into sentences. The MAP framework exposes these weaknesses.
- Reference architectures provided by RSE and Perspective are largely reminiscent of OO systems. Catalysis focuses on architectural patterns, whilst KobrA discusses at length technical issues related to composition of component hierarchies.

- Both RSE and Perspective have SDP with reasonably good coverage and control mechanisms. Treatment of SDP in Catalysis and KobrA is rather general.

These evaluation results confirm the first hypothesis of this research that the theoretical basis of the existing CBSD methods is weak. This is particularly the case with system modelling.

Chapter Five

The Proposed CBSD Method
NAVITA – An Introduction

5.1 Introduction

This chapter provides an overview of the proposed holistic approach CBSD, NAVITA. It is holistic in a sense that it addresses all three important aspects of software development, namely, System Modelling, Software Architecture and System Development Process (SDP), in a true component-based fashion. The name NAVITA is derived from an abbreviation of two Sanskrit words "Naviinamh" meaning new and "GhaTaka" meaning component (Sanskrit Dictionary, n. d.).

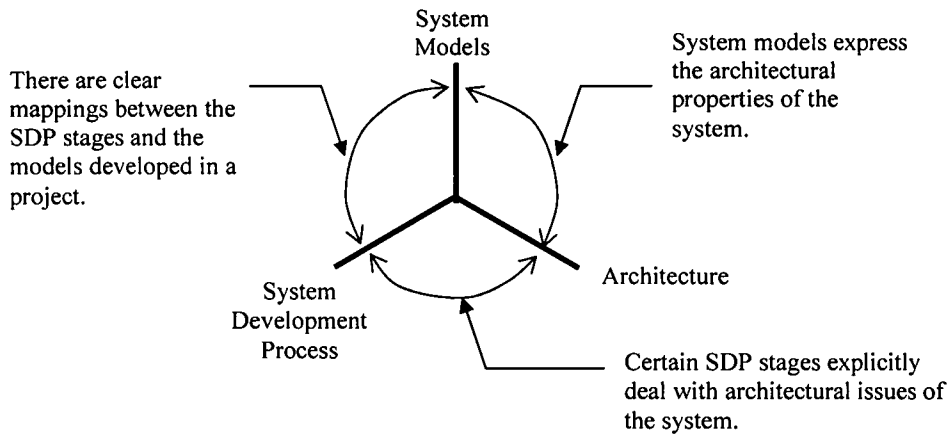
Being a component-based method, NAVITA is fully geared towards the realisation of the "reuse first" vision of software development. In this approach, once a general understanding of the users' requirements is established, the search for an existing application that satisfies the requirements begins. If the matching application is not found or reuse of it not viable, the search looks for an application that could be adapted. In many cases, users may be encouraged to choose an existing application if the gap between their expectations and the available application is not wide. If not successful, the system is decomposed and smaller components that can be reusable in the application are examined. This process may be repeated many times. Only those components that are neither available nor viable for reuse are to be analysed in detail, designed, implemented and tested. This reuse philosophy is not treated as a fad but a strong principle appropriately manifest in all three main aspects of NAVITA. SDP of NAVITA encompasses a spectrum of development scenarios of the CBSD approach, from acquisition of an entire matching application in which very little system modelling and development activities are necessary, to development of an application for which no components are viable/available, and

need to be created from scratch. System Modelling activities are tied to SDP stages; therefore, only necessary models are produced. Software Architecture of NAVITA prominently features the quality of 'pluggability', which allows parts of the systems to be added, removed and upgraded on the go, facilitating reuse of different kinds of component.

5.2 Orthogonal View of NAVITA

The three elements of NAVITA, System Modelling, Software Architecture and SDP are tightly linked to each other. Figure 5.1 shows general correlations between those elements, which are described in greater detail as the discussion progresses.

Figure 5.1 Orthogonal View of NAVITA

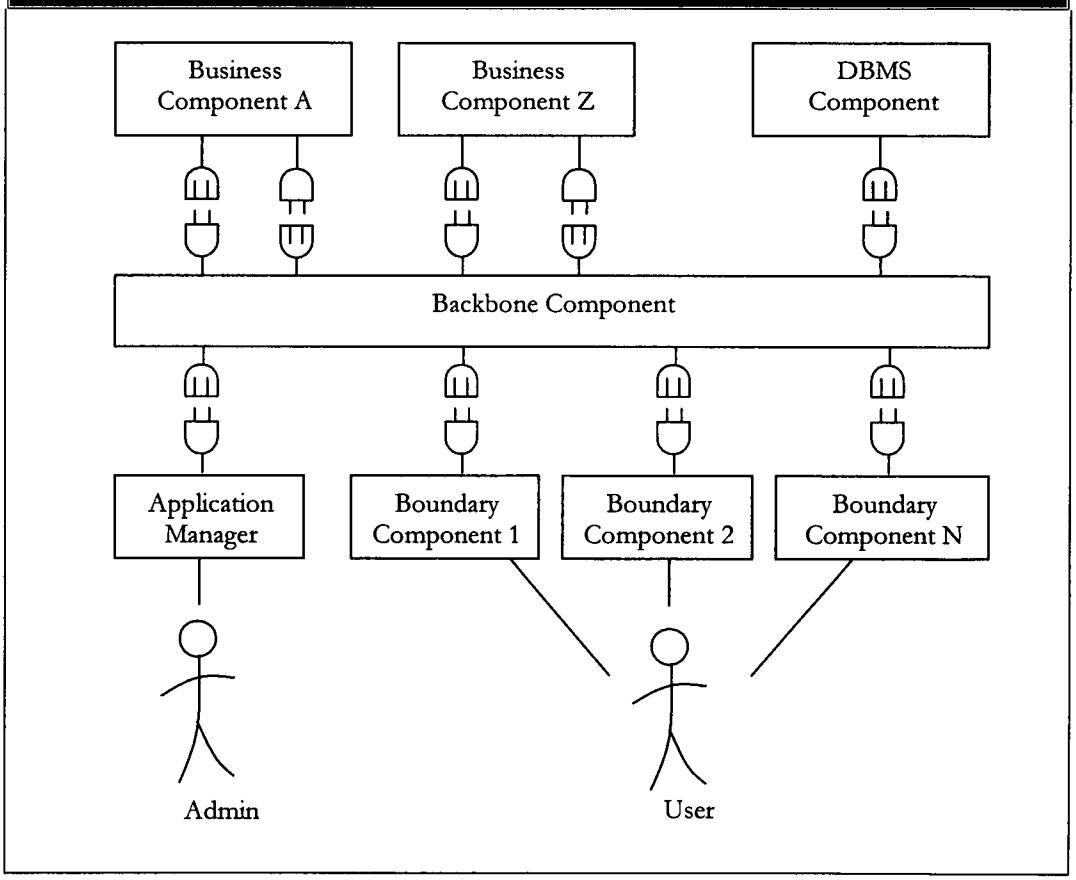


5.2.1 NAVITA Software Architecture

NAVITA provides the following reference architectural model that gives a realistic vision of software architecture as necessitated by CBSD and modern software technologies. In this architecture, every 'application', or an assemblage of collaborating component, has three key elements: a Backbone component; an Application Manager component; and business, boundary and other generic components, such as Database Management System (Figure 5.2). Backbone and Application Manager components are the most essential ones, even an application without any user functionality will have them. To add functionality to the system, the application administrator will use the Application Manager component to first define services with the Backbone component. This involves naming of the service and the operation signatures of the operations used for this service. This registry of

services and operations are managed by the Backbone component. The defined services are logical in a sense that it does say how physical components ought to be composed. Once services are defined, the application administrator can then add physical components by specifying the service(s) the component offers and requires, represented by plug and socket icons. A boundary component generally uses (from business components) one service only, while physical business components may provide and use multiple services from multiple components. At runtime, the Backbone component ensures that components communicate with each other with no knowledge of their physical locations, design paradigm or implementation technology.

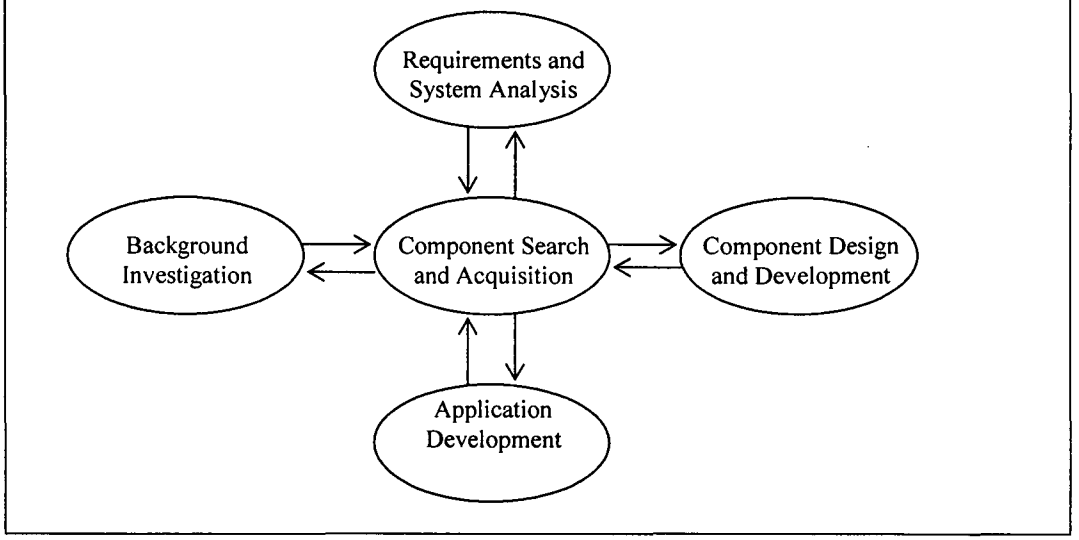
Figure 5.2 NAVITA Reference Architecture



5.2.2 NAVITA System Development Process (SDP)

The SDP envisaged by NAVITA (Figure 5.3) reflects the basic nature of system development according to the "reuse first" principle of the component-based approach.

Figure 5.3 SDP of NAVITA



System development with NAVITA starts with a Background Investigation into the business processes, project feasibility – in terms of time, technology, organisational, costs, benefits, and reuse of existing components – and a general outline of the users' requirements. Once the feasibility of the project and requirements for a new system are established, existing applications are investigated for potential reuse. If applications are not found in the Component Search and Acquisition stage, their reuse unviable or users' requirements cannot be adjusted to an available application, the requirements are further analysed, helped by user interface prototyping. Then specifications of the system's main logical components are produced in the Requirements and System Analysis stage. Then, components for these specifications are searched again. If existing components found to be reusable, they are certified. If components need to be developed or tuned, the development moves into the Component Design and Development stage. Once the components are developed, tuned and/or ready to be reusable, application is assembled, tested and delivered to the users, who then accept it. Therefore, system development with NAVITA is based around the central activity searching and acquiring relevant existing components. This whole SDP is often carried out in an iterative and incremental fashion.

Chapter 7 discusses the NAVITA SDP in detail.

5.2.3 NAVITA System Modelling

In line with the lessons learnt from the evaluation of the existing CBSD methods, NAVITA provides a set of global and contextual models that would enable analysts to model important characteristics of the systems, as well as ensure that the global models are complete and consistent, both internally and externally (see Section 3.4.1). Furthermore, being a component-based method, NAVITA suggests modelling only when necessary, i.e. depending on the development scenario. Models are tied to SDP stages, and the SDP stages required in a project are dictated by the development scenarios. For example, if a matching or similar application is found for the users' requirements, little or no modelling is carried out. The following are the main models used in NAVITA:

- **Business Analysis** – NAVITA Modelling starts with Business Process Modelling (BPM) with an emphasis on gaining an understanding of the business domain, the users' requirements, system boundary and necessary improvement to the business processes. This thesis will not cover the business process modelling due to limitations of the research scope and thesis space. Broadly speaking, NAVITA suggests using the standard business process modelling language (BPMN) (White, n. d.; BPMI, 2004) with the modelling techniques suggested by SELECT Perspective (Allen and Frost, 1998). As in SELECT Perspective, two main diagrams are used.
  - **Process Hierarchy Diagram** – This diagram shows a simple hierarchical breakdown of the business processes.
  - **Process Thread Diagram** – This diagram shows the order in which Elementary Business Processes from the Process Hierarchy Diagram are executed.
- **Requirements/System Functionality Modelling** – Functionality Model is usually developed in conjunction with the BPM. The latter emphasises the context in which the new system will operate, while the former underlines what the system would do. In this modelling, the following diagrams are used to capture the users' requirements from three perspectives: process, data and interaction.
  - **Context Diagram** – This diagram provides an overview of the system with emphasis on the roles of users, their interactions with the system, and manual and computerised processes of the system. Although there

are similarities between the NAVITA context diagram and traditional context diagrams, NAVITA diagram is more expressive. This diagram is also used to explore the often grey line between computerised and manual processes by differentiating between two types of boundaries: traditional system boundary and 'greater IS boundary'. The former represents the boundary of the computerised system, while the latter represents the Information System as a whole including manual processes. This separation allows analysts to investigate the system from a much broader perspective. NAVITA context diagram is discussed in Chapter 8.

- **Middle-level Functionality Diagram (MFD)** – This Middle-level System Functionality diagram is the main functionality diagram, which is supported by some other diagrams and documentation. The main purpose of this diagram is to show, in a clear and simple way, the main functional requirements of the system, while the textual documentation gives further details of these functional requirements as well as captures non-functional requirements (Chapter 9)
- **Lower-level Functionality Diagram** – The granularity of the functionality units is fixed at the middle level functionality modelling, and detailed analyses of the complex functionality units are done using Lower-level Functionality diagrams. A LFD is a UML Activity Diagram-like diagram containing activities, the flows of information and interactions between actors. (Chapter 9)
- **System Interaction Modelling** – This modelling is made up of Logical Screen Layout (LSL) and User-System Dialogue Model (USDM). A LSL shows the logical and static interface between the system and the user. This is a simple and largely informal diagram used to model the input and output data of each functionality unit. It is not at all concerned with Graphical User Interface (GUI) and navigational issues. LSL modelling attempts to visualise the static interface of the system, and in USDM, detailed dynamic interactions between the system and the user are explored. In a sense, USDM is Logical Screen Layout Diagram with the added time dimension. This diagram provides an important crosschecking between the functionality model (through its steps) and

the LSL (through its data items). A combination of these three diagrams provides a complete 'external' view of the system functionality. LSL and USDM are described in Chapter 10.

- **Information Model** – Information Model is used in a number of stages. The main aim, however, is the same: to show the data structure of a group of data items, in terms of entities/classes and their relationships. Chapter 11 explains the concepts and modelling technique of this model.
- **FEM** – FEM shows the relationships between functionality units and entities/classes. From Middle-level Functionality Model and USDM, events are identified, and from the protocol model, various effects are extracted. These effects are mapped onto entities from the Information Model. This matrix, therefore, brings together the main models, and helps ensure consistency and completeness of these models. This modelling is discussed in Chapter 11.

⇨ **Architectural Analysis** and **Component Design**

- **Protocol Model** – For each functionality unit, a protocol model is produced to show how a logical boundary Component and a logical Business Component will communicate to realise a functionality unit. With Logical Screen Layout and User-System Dialogue Model, analysts are able to analyse in detail the interactions between the user and the system. Now, the interactions between the boundary Component and Business Component, to provide the required service to the user, are analysed.
- **Logical Component Specification** – This specification, a culmination of a number of models produced so far, precisely defines the interface of the logical components with the system. The specifications cover the functional, information and interaction aspects of the components. These specifications are the basis for the search of existing components, and possible design and implementation. Protocol Model and Logical Component Specification are described in Chapter 12.
- **Component Design** – Development of components either from nought or existing similar component starts with internal design of the components. The specifications of services in the previous modelling are logical in a sense that they do not tell us how actual components should

be designed. It is at this stage that analysts determine how best to create required components using a certain existing implementation technology of choice. This diagram shows the architectural design of the system in terms of its physical components. The main issue here is how best to create physical components that are not only compliant with service specifications, but also architecturally sound and reusable. This modelling is very important for CBSD methods and NAVITA provides detailed discussions on issues surrounding this modelling. Component modelling is described in Chapter 13.

- **Sequence Diagram**, **State Transition Diagram**, and **Deployment Diagram** – These UML diagram are also used in appropriate contexts: Sequence Diagrams to model interactions between objects within a component, and between multiple components; State Transition Diagrams to model event-driven lifecycle of objects and components; and Deployment Diagrams to show allocation of software components to hardware devices. Sequence Diagram and State Transition Diagram are discussed in Chapter 13. Since the deployment modelling is relatively simple, for example see (Bennett et al, 2001), it will not be repeated in this thesis.

Figure 5.4 shows the general flow of development of NAVITA models, dependencies between diagrams and other documentations used in those models, and various crosschecks between them. Single-headed arrows in the figure indicate the timing of development and/or one way dependency of elements of the models, while double-headed arrows show interdependencies between them. It is worth reminding that if a diagram is dependent on another, every time the first diagram is revised, the second diagram needs revision too. If more than one arrow converges in a diagram or documentation, the diagram or documentation is a crosscheck between diagrams from which the arrows originate.

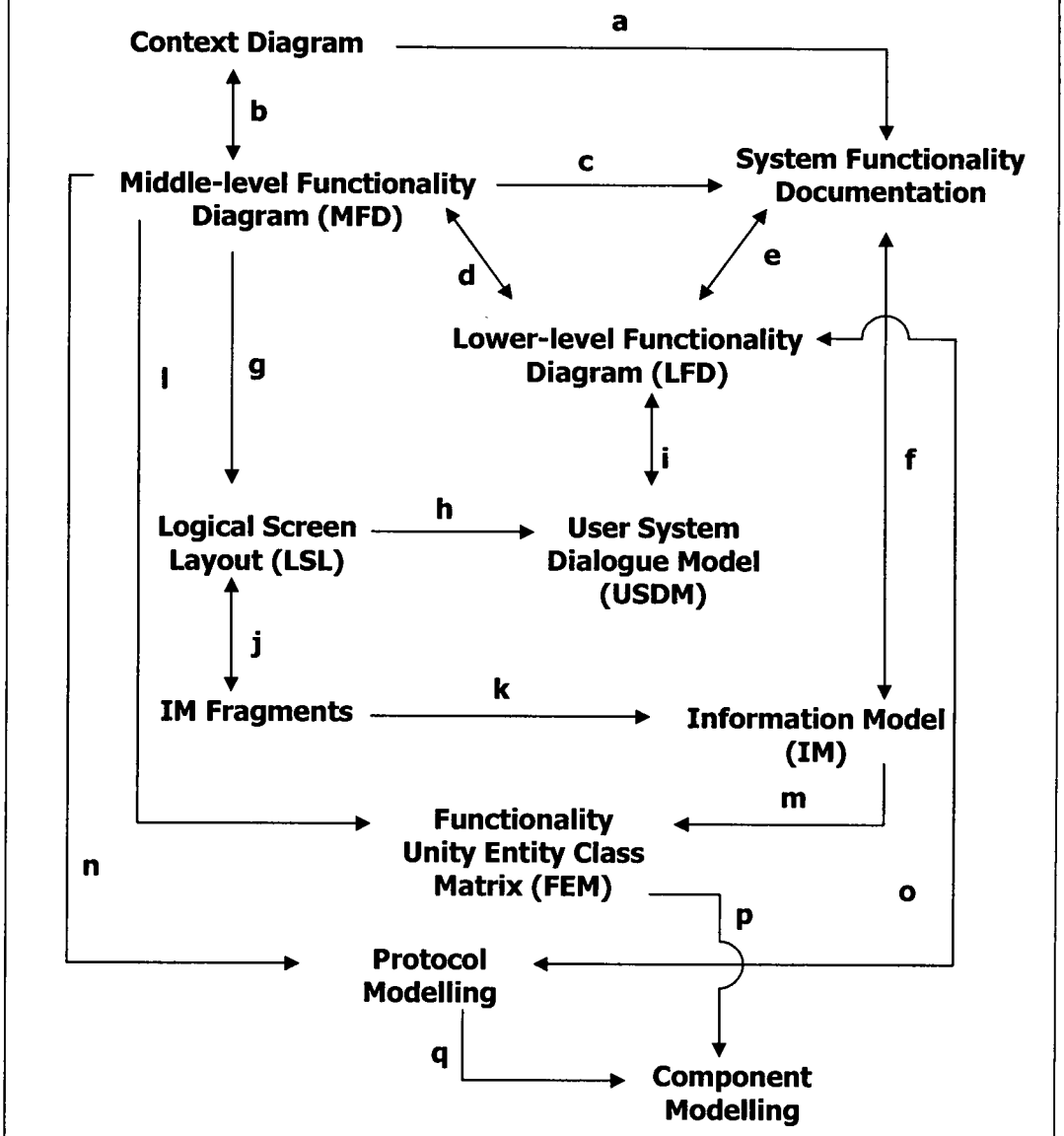
a. Actors, interactions and system boundaries from the context diagram are documented in the system functionality documentation

b. Actors, interactions and system boundaries from the context diagram are brought forward into the MFD

c. Descriptions of functionality units in the MFD are documented in the system functionality documentation

d. Each complex functionality unit analysed in detail using LFD by breaking it down into smaller-grained activities. This analysis can often lead to revision of original MFD in terms of changes to the boundaries, limits of the functionality and even actor(s) involved. Therefore, new and detailed knowledge gained from LFD feeds back into the MFD. As noted, any changes to MFD are likely lead to changes in the documentation according to the flow c.

Figure 5.4 Flow, Dependency and Crosschecks between NAVITA Models



e. Descriptions of the functionality units in the documentation feed into the LFD. A combined flow d and e into LFD enables it to become a crosscheck between the MFD and the documentation, because it enables the analyst to approach this detailed analysis from two independent angles – first from informal textual

description of the functionality unit focusing on what the process entails, then a diagrammatic analysis showing how the user would interact and system would work – and check one against the other. In particular, it ensures that the steps in the descriptions of functionality units are consistent with the activities in the LFD.

f. The initial IM bases itself from candidates for entity classes identified from the documentation. After crosschecking with IM fragments from LSL, it may lead to revision of what entity classes are inside the system and what are not. This may often lead to revision of the way functionality units are described.

g. Details of interactions, carried from the context diagram through MFD and crosschecked in LFD, are used for the analysis of input and output data between the users and the system. For each functionality unit in MFD, a LSL is produced.

h. Based on the input and output data in LSL, USDMs are produced, which bring together IO data from LFD and IO data from LSL, enabling a crosscheck between the two diagrams. Furthermore, the control structures of the two diagrams can be crosschecked as well.

i. IO Data in LSL are analysed to form entity classes and relationships. In some cases, it may lead to identification of missing data and hence revision of the LSL.

j. IM fragments from LSL are then used to crosscheck the original IM, which often requires a revision in the light of the concrete analysis of data.

k. Functionality units are brought forward into FEM.

l. Entity classes are carried into FEM that provides a crosscheck between the two models.

m. Entities from the IM are fed into FEM.

n. For each functionality unit a protocol model is produced; the control structure derived from the protocol analysis can be used to crosscheck the control structure in LFD and USDM too.

o. Control structures are brought forward for crosschecking.

p. Entities and functionality units provide the basis for NAVITA component modelling.

q. Operations identified in the protocol analysis are allocated to classes and components.

5.3 NAVITA Filters: Conditions of use

NAVITA is suitable for use in development of software applications with the following characteristics.

- Database-driven business/enterprise applications with some complex computational processes: modelling techniques and software architecture proposed by NAVITA assume that the applications are typically business information systems. This method provides no methodological guidance on, for example, how to develop a compiler or a game application using the CBSD approach.
- Interactions with users through GUI: modelling techniques and software architecture of NAVITA also assume that users interact with the system directly or through other human actors. System analysis in NAVITA depends heavily on the analysis of the user-system interactions.
- Medium-sized applications: all three elements of NAVITA assume that the applications have between 10 and 50 complex functionality units. Applications with fewer functional units may not benefit much from this method, whilst applications with substantially more functionality units may require more detailed methodological guidance on issues such as implementation, testing, documentation and project management.
- Time, safety and dependability are not of primary concern: NAVITA does not provide methodological guidance, such as the use of formal specification techniques, for the analysis and design of applications such as real-time systems.

Having said this, parts of NAVITA may be adapted for use in development of applications without these characteristics; for example, the NAVITA protocol model may be used for analysing real-time applications.

Chapter Six

NAVITA Software Architecture

6.1 Introduction

In this chapter, the architectural model suggested by NAVITA is discussed in detail. The role of architecture is extremely important in component-based software applications, and NAVITA puts software architecture on an equal footing with the system development process and system modelling. The main reason is that component-based applications need to be built so that it will be easy to add, remove and upgrade parts of the system with minimal ramifications on the rest of the system. A good software architectural model helps analysts achieve qualities such as flexibility and reusability in their designs (Crnkovic and Larsson, 2002). In this method, analysts first attempt to understand the users' requirements. If some or all parts of the system need to be developed, due to unavailability or non-viability of reusing the entire application, then requirements must be translated into a logical architectural design. The application architecture is defined in terms of services the components should offer. Specifications of these services provide the basis for the acquisition, or if necessary eventual development, of the components. Therefore, application architecture is the key to most development stages.

6.1.1 Software Architectural Models Suggested by Existing CBSD Methods

Software architectural models suggested by existing CBSD methods are examined in the survey paper in Appendix II. Models suggested by various existing CBSD methods tend to be rather generic. For example, RSE and SELECT Perspective promote the three-tier architectural model, which is suitable for many applications including non-component-based ones. Such genericity is unhelpful where analysts

are interested in a more specific ways to break down the system into components. NAVITA Software Reference Architecture is formulated in such as way that it is generic enough to be applicable to a range of systems, while restricted to only component-based applications.

6.1.2 Software Architectural Model Envisaged by NAVITA

NAVITA Software Architectural Model, in line with the philosophy of CBSD, does away with the notion of application as a monolithic entity in terms of the way the system is implemented and is seen by its users. In its place comes the notion of a union of loose and fairly independent components collaborating with each other through a common 'platform', onto which components can easily be added and removed in order to keep the system in line with changing user requirements. In this architectural vision, components are added to the system if and when new capabilities are required and removed if they are deemed unnecessary. NAVITA Software Architectural Model is inspired by a range of modern software technologies.

6.1.3 Enabling Technologies

Middleware architecture and technologies such as CORBA, Java RMI, DCOM, OLE, ActiveX (Szyperski, 1998; Orfali and Harkey, 1997) provide mechanisms for components to share their services across applications and implementation environments. Programming languages such as Java allow programmers to create self-contained components (Sun, n d.), while commercial application development environments such as Visual Basic enable developers to reuse components visually (Microsoft, n d.). Most commercial web browsers such as FireFox (Mozilla, n. d.) provide plug-in managers that allow users to update and maintain components in the browser applications on the go.

6.1.4 Hardware Analogy

Many authors on CBSD methods are quick to highlight the analogy between the standard computer hardware engineering approach and the CBSD software development approach, citing the importance of modularity of components, separation and standardisation of component interface, and pluggability of components. NAVITA software architecture draws a deeper analogy with the

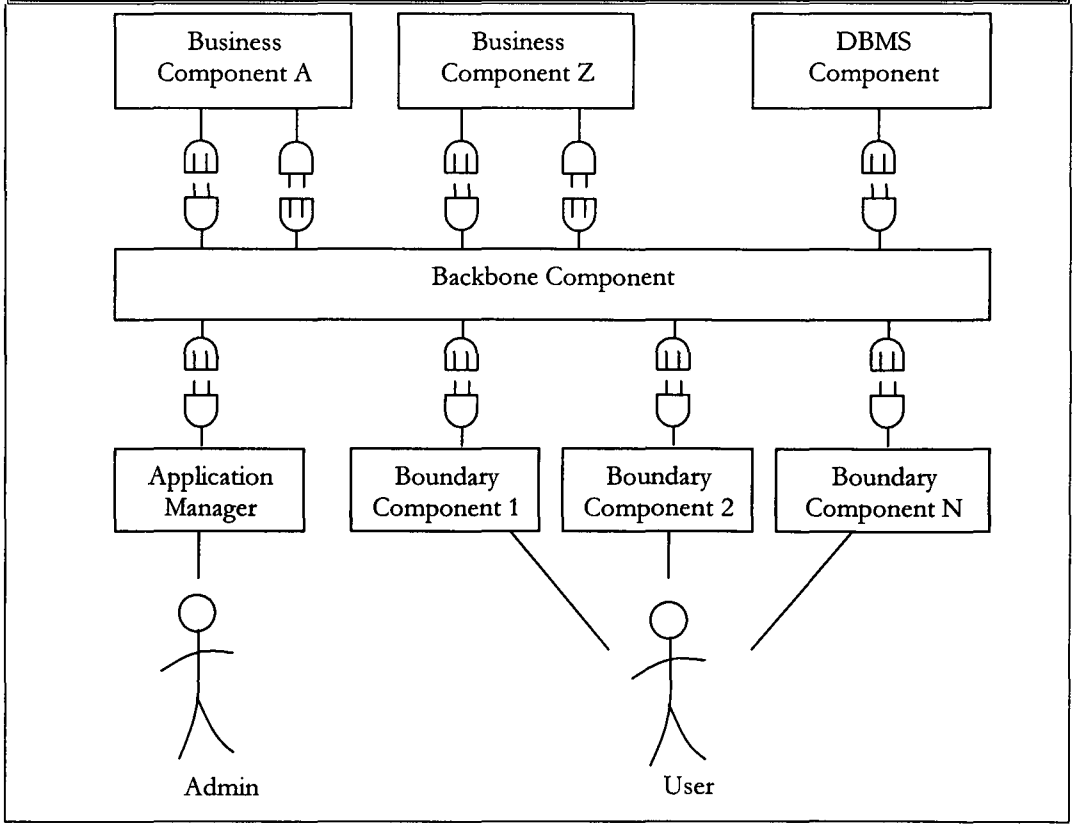
computer hardware. One of the key aspects of computer hardware architecture is the use of a core component into which other components plug, namely, the "motherboard". The motherboard functions as the central component that defines the core of the system by way of expressing the interfaces with other components in the system, effectively defining what components can and cannot plug into it. The use of the motherboard component provides a simple, flexible and effective computer hardware architecture. NAVITA Reference Architecture suggests the use of such a component in software applications too.

6.2 NAVITA Reference Architecture

NAVITA Reference Architecture is based on three key properties:

- the Backbone component provides the platform for component communication
- the use of Application Manager in each application serves as an accessible mechanism for addition and removal of components from the application
- the concept of Service is used to describe the functional behaviour of components

Figure 6.1 NAVITA Reference Architecture



6.2.1 The Backbone Component

The Backbone component is the central component through which services of components are provided and used. In this respect, it is the most important component in a system; without it other components could not function together as a system. The Backbone component provides a registry and serves as a medium of communication for all other components. Each application has a Backbone component built into it. Everything in this architectural model is a component including the Backbone component; in a sense, it is a true component-based architectural model. The Backbone component primarily deals with the following tasks:

- maintaining the service registry, which holds detailed information about the services and interfaces of all components
- keeping the registry updated when a new service is defined, an existing service is removed, when a component that satisfies some service(s) is installed, and when a component is removed
- during the runtime, resolving the requests for component services in a way that is platform-independent and location-transparent

When a service is defined with the Backbone component, it will generate stubs for the client and the server, i.e. interfaces that the real client and server must implement in order to be able to communicate with each other (Orfali and Harkey, 1997). Once defined, implementation of these interfaces, or other compatible interfaces, can be registered with the Backbone component. The Backbone component will have to ensure that the expected interface matches the actual interface of the component. At runtime, when client components invoke services from the server components, the Backbone component ensures that the components are identified and their services used, irrespective of the physical location of the components involved. For other key services typically provided by Object Request Brokers, see Szyperski (1998).

The Backbone component therefore acts like a middleware architecture, and it has the following key properties:

- platform-independent – implementation technology of components do not present a barrier to the integration of components in an application
- location-transparent – the physical location of the components, which may be the same computing device or device across a network, does not present a barrier to the way in which services of components are

provided and used. Simply put, components do not need to know each others' physical locations in order to share their services.

- paradigm-neutral – components may be designed and implemented using any 'paradigm' such as Structured, OO, Relational
- flexible openness – this architecture supports both open and closed architectural models. An open architectural model is a model in which the kinds of components that may be added to the application are not limited by the Backbone and Application Manager components; therefore the exact services are not 'hardwired' into the application. With the closed architectural model, the services of the application are restricted, mainly by limiting the service definition functionality of the Application Manager and Backbone components.

6.2.2 Application Administrator

An application administrator is someone who is in charge of managing the components in an application. The administrator role can be played by some of the users, or a dedicated individual or group of people, if the application is large and spans multiple locations.

6.2.3 Application Manager

In each application, there will be a specialist boundary component, known as the Application Manager component. This is used by Application Administrator to interact with the Backbone component in order to perform component administration tasks such as registering components with the Backbone component (see Section 6.2.1).

6.2.4 Service

A service is a contribution made by a component of any type, such as boundary, business and DBMS, to realise a functionality unit. In many cases, a service is a set of publicly accessible operations provided by a component, often involving a complex protocol for communication, such as that between a desktop application and a printer manager.

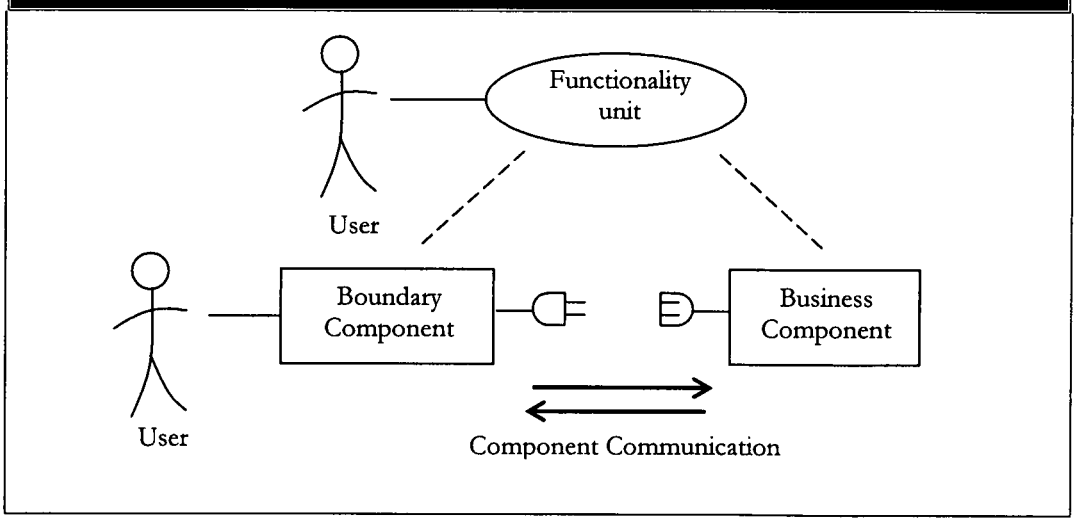
6.2.5 Logical and Physical Architecture

NAVITA Software Architecture is expressed at two levels of abstraction: one logical and one physical. Logical software architecture is mainly concerned with the high-level (system-level) description of the system in terms of services provided by its components in a way that is not influenced by considerations for their implementation. Physical architecture is more focused on lower-level (component-level) design of the individual components.

6.3 Logical Architecture

NAVITA logical architecture assumes that a functionality unit (Section 9.3.3) is realised by collaboration between two logical components: a boundary component through which users interact with the system and a business component that encapsulates the business logic, constraints, process/procedures and data. The two components are bound by a contract, and their interactions realise the functionality unit.

Figure 6.2 Boundary component and logical business component realising a functionality unit



6.3.1 Logical Boundary Component

A logical boundary component is an encapsulation of the mechanism through which an actor interacts with the system while using a functionality unit. In many situations, it is a set of Graphical User Interface (GUI) objects, using which a human actor interacts with the system. The primary concern of the analysts at this logical level is not how these boundary components are made up of various visual user

interface objects, or in cases of interaction with other systems, the exact nature of communication protocols, but rather the nature of logical interactions with the user in terms of inputs and outputs, and communication with a business component.

6.3.2 Logical Business Component

A logical business component provides a complete service to the logical boundary component to realise a particular functionality unit. It is neither good nor feasible to implement a business component for every functionality unit. However, logical architectural analysis is not about identifying physical components of the application. Rather, it is about identifying and specifying the interfaces of business components providing certain functionality units in a way that does not indicate how the application should be composed. Therefore, the main aim of logical architectural analysis is to explore the externally visible interfaces and communications of business components by first separating out the user interface part of the system and then envisaging how the boundary will have to respond to the needs of the users. In this architectural analysis it is assumed, for simplicity, that the Backbone component is not involved in the communication. Chapter 12 provides further discussion of logical architectural analysis suggested by NAVITA.

6.3.3 Logical Component Communication

Components communicate through service operations. These operations need not be simple operation calls, such as those in object collaborations (Jacobson et al, 1992). Instead they may involve complex exchange of messages, such as those between message-based synchronised processes in real-time applications (Burns and Wellings, 2001). In a typical client-server configuration, the server does not need to 'know' its clients; only the client needs to know its server (Edelstein, 1994). Therefore, clients do not have to register with the server in order to use its services, except in special circumstances, for example, for security reasons.

6.4 Physical Architecture

While the logical architecture defines the functionality of the application in terms of services of logical components, useful for the purpose of defining what components should do and their interfaces should be, it is not a pragmatic design for implementation. Some services may be related and always used together; in such

cases these services can be provided by a single physical component, reducing the number of components and complexity to manage.

Physical components, like logical components, may be of two main types: boundary components and business components. Other components such as Database components, language (library) components, lower-level service components such as sort, search and queue components can also be accommodated in this architecture, as discussed in Section 6.3.

6.4.1 Physical Boundary Component

These components are compositions of GUI or other interface objects, through which users communicate with the system. They mainly communicate with the Backbone component to acquire the services they need. Related boundary components can be composed together. See Chapter 13 for further discussion.

6.4.2 Physical Business Component

Various definitions of component by existing CBSD methods are presented in Appendix II. The survey shows that different authors emphasise different aspects of components: some emphasise the technological aspect (Szyperski, 1998), some regard it as essentially non-technological (Jacobson, 1997), some stress the relationships between components and business processes (SCIPIO), some treat it as a generic application (Atkinson et al, 2002), and so on. Not only CBSD methods, but also non-CBSD methods have their own notions of the term "component".

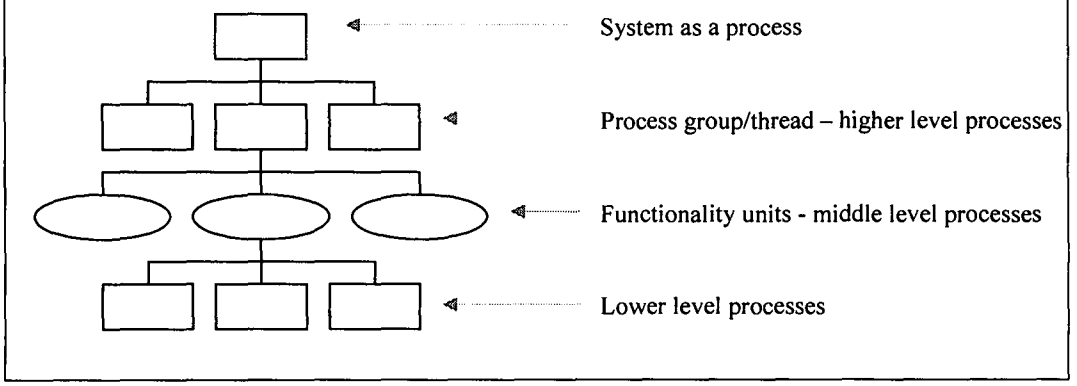
6.4.2.1 Fundamental approaches to decomposition of system

Detailed analysis of a system always requires systematic decomposition of the system by means of identifying its constituents and their dependencies (Wieringa, 1998). The nature of a system's constituents – dictated by the paradigm of the method – affects the way in which the system is decomposed. There are two fundamental main perceptions of the nature of a system's constituents: one promotes the process-centric view and the other, the structure-centric view.

6.4.2.2 Component as a process or group of processes

In this view, the system and its constituents are all processes by nature. The system, as shown in Figure 6.3, is often described as a hierarchy of processes. In such a hierarchy, the system is represented as a single process at the top of the hierarchy which is repeatedly decomposed until further breakdown is deemed unnecessary.

Figure 6.3 Component as a process or group of processes



If a component is treated only in terms of pure processes, then there are three main possibilities for granularity of components:

- **Component as a Low-level Process** – component as a unit smaller than that of a functional process is untenable because it is unlikely to yield a high amount of reuse.
- **Component as a Middle-level Process** – component as a functional process is desirable only so long as it is behaviourally rich, complex and worthwhile. It however would not represent a step forward from the traditional reuse approach.
- **Component as a High-level Process** – component as a sub-system or an application itself is an attractive prospect from the reuse point of view.

Then the question is this: How can analysts compose process into larger ones or components? There are a few notable approaches to answering this question.

In methods such as SSADM, composition of smaller processes into larger ones is generally arbitrary since there is no principled way to ascertain what processes make up a higher-level process or component. The usual approach is to break up the system processes along departmental lines, such as Order Processing and Payroll. However, analysts are at liberty to decide whatever forms of sub-system there

should be in the system. Such a random composition of components cannot lead to a stable basis for component composition.

Alternatively, processes that are used together can be grouped together. The emphasis is therefore upon how the users will be using the processes. For example, in the case of library system, all functionality units used by readers are put into a component, those used by librarians into another component and so on. There are a number of problems with this approach: first, functionality units are often shared by many users such as the 'search' functionality (in that case we may decide to have a separate component for share functionality units), and second, often actors who use certain functionality units may change over time. For example, Borrow Book may be once used by a librarian, then a reader might use it and later both. It is clear that this approach does not provide a stable basis for grouping processes.

A more sophisticated approach, which can be called the 'business-driven approach', would be to group together processes that are chained together by their input and output values (Allen and Frost, 1998). For example, for Borrow Book, Return Book will be necessary, for Make Reservation, Cancel Reservation and so on. This approach goes some way to solve the question; however, since all functionality units are related in some way, it is difficult to determine where to draw the line.

6.4.2.3 Component as an encapsulation of data structure

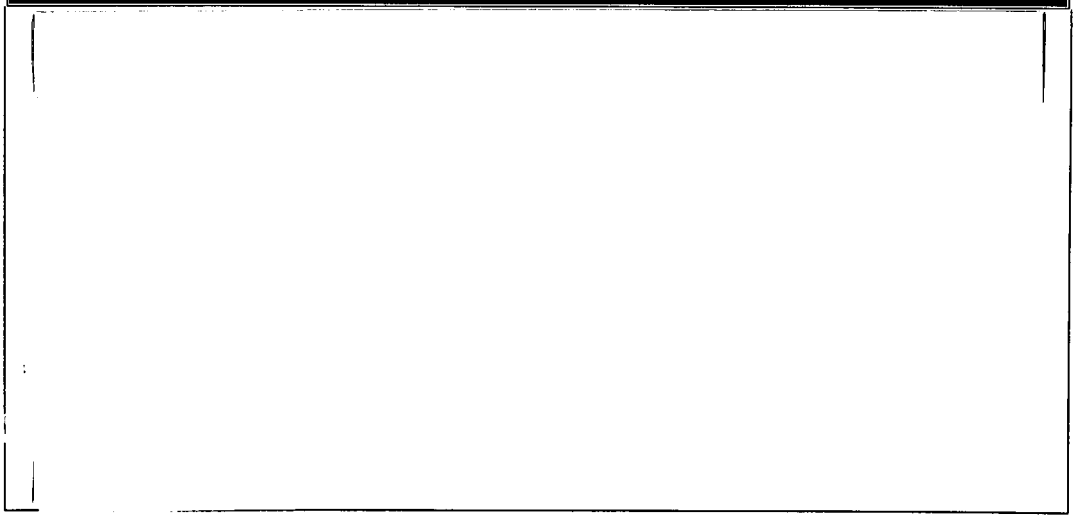
In this perception of the system, constituents of a system encapsulate a certain part of the data structure. Components are created by accommodating operations to the underlying data structure. Manifestation of this idea is clearly seen in Abstract Data Types (Watt and Brown, 2001). Each component in this perception is centred on a well-defined data structure with a finite set of operations. Data structures such as Stack, List, and Queue are neat components because their data structures are completely encapsulated by their operations. These components are self-contained and cohesive, making them highly versatile and their reuse convenient. Object-orientation can be explained as an attempt to apply this idea universally. Like ADTs, OO classes have internal data structures which are encapsulated by a set of operations. There are other similar approaches such as JSD entities (Cameron, 1989) and high-level processes in Yourdon Structured approach (Yourdon, 1989).

When entities of business information systems are taken as a basis for objects, difficulties arise. Entities of information systems are not as self-contained as stacks or lists, and they have complex and large data structures. Only if classes are as reusable as ADTs, would there be a perfect means of reuse.

6.4.2.4 Difference – The Problem

The fundamental differences between these two approaches reflect the two perspectives analysts have on components: one business and one technical. From the business point of view, it is more convenient to think of components in terms of processes (Casson, 2000; Allen and Frost, 1998). This view of component does not take into account technical and architectural issues of integrating foreign components into an unfamiliar system. On the other hand, the technical view would favour the structural approach because it enforces the integrity and cohesiveness of the components. In doing so, it is easy to ignore the business perspective of the component.

Figure 6.4 How process-oriented and structure-oriented designs accommodate different kinds of changes (Jacobson et al, 1992)



Jacobson et al (1992) show that, contrary popular belief, OO software is more maintainable than Structured software, and that OO approach is better only so long as the nature of change is structural. In Figure 6.4 for example, addition of a new account type such as student account will not affect the rest of the class hierarchy, while adding a new operation such as to calculate interest will affect all classes. On the other hand, adding a new account type will affect all processes that access the account data store, while adding a new account operation will have no impact on

both data store and processes. Therefore, these two approaches are not only significantly different, but also paradoxical.

NAVITA argues that there are two problems with the traditional understanding of classes:

- ⇨ Granularity of operations tends to be very low. By focusing on attributes of individual entities, operations allocated to entities become too stretched over classes. Therefore, classes can no longer give a sense of what they do in business terms.
- ⇨ The previous problem is partly caused by the nature of relationships between classes which tend to get mixed up. Analysts must distinguish between relationships that deal with data structure and relationships that represent 'interactions' between components.

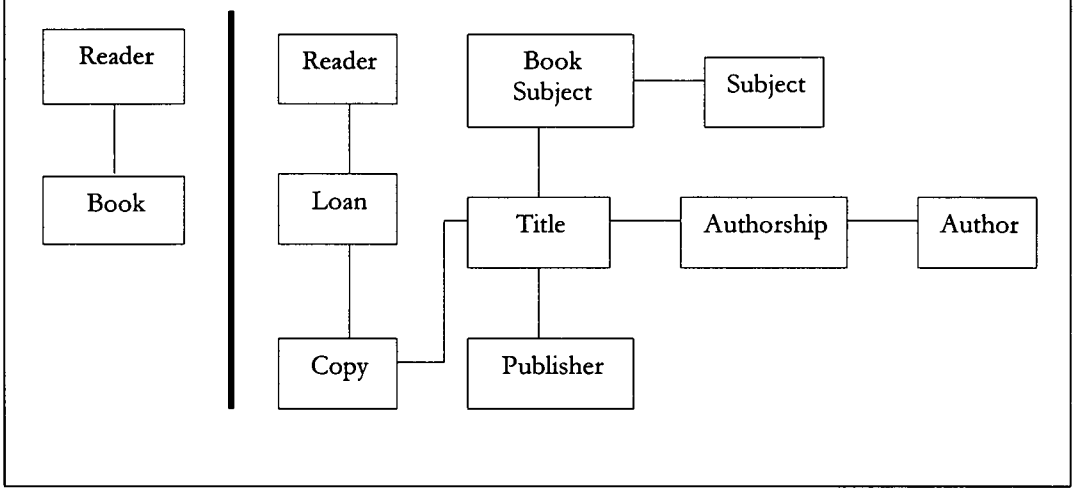
6.4.2.5 Granularity

A cornerstone of OO is arguably the unification of data and process. Instead of regarding systems as being composed of separate data and process parts, OO seeks to combine data and process into coherent objects – with data in their cores and processing surrounding them (Booch, 1991 and 1994). Objects also closely reflect real-life entities and hence it is said to become much more 'natural' and 'obvious' to think of systems as being made up of collaborating objects. The concept of "object" seems to work well with objects in application where there is a neat encapsulation of data by operations. OO works well with application such as GUI systems (Shneiderman and Plaisant, 2004). When the concepts are imported into the spheres of Information System, certain adaptations are made in order to accommodate the data-oriented nature of the application. Information Systems are largely database-driven and hence class model of a typical IS has a strong RDB feel and connotation. Moreover, operations in those classes are so low-grained that it is difficult to establish what the system might be doing by studying class operations alone, because these operations tend not to have much resemblance to business operations. Such diagrams tend to show both data structure and operations and it is hard to find genuine integration of data and processing in these cases. For example, this has led Jacobson et al (1992) to invent 'control' objects that absorb operations that will not naturally sit with "entity objects".

6.4.2.6 Class Relationships

The problem of low granularity of operations is often caused by the failure to distinguish between the nature of class relationships and data structure relationships. Classes need to represent substantial entities with rich behaviour and a certain amount of independence. Instead of objects representing real-life entities, class models are usually laden with details about the structure of information. To illustrate the problem let us study the following entity/class diagram for a typical library system.

Figure 6.5 Single entity and entities with strong relationships



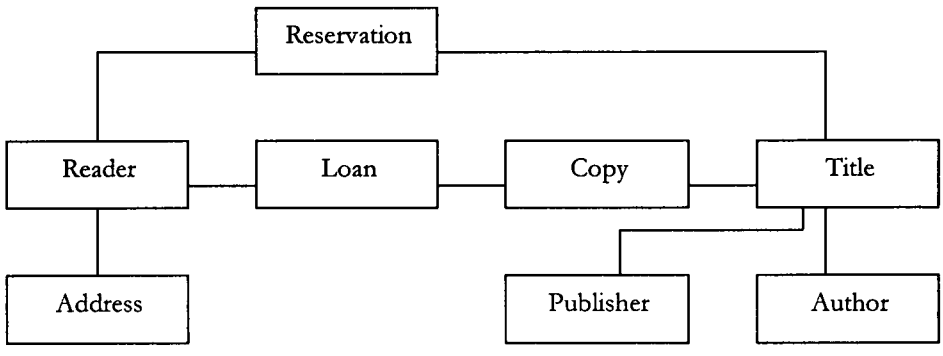
In the first diagram, there are two entities/classes and a relationship and in the second diagram, there are substantially more of both. However, in real terms, they both represent the same things.

There are two important things to note here. First, even though many new entities are introduced in the second diagram, these new entities simply elaborate the data structure, and do not add anything unaccounted for by the first diagram. These new entities are created for technical necessity, such as to satisfy Relational Data Analysis constraints (Goodland and Slater, 1995). Second, the relationship between Reader and Book in the first diagram represents real-life interactions such as borrowing books, but relationships between say, Copy and Title do not. In a system therefore, there are entities and relationships that are clearly created to represent the data structure, and there are entities and relationships that are created to represent important entities and their relationships in the 'real' world.

6.4.2.7 NAVITA Solution

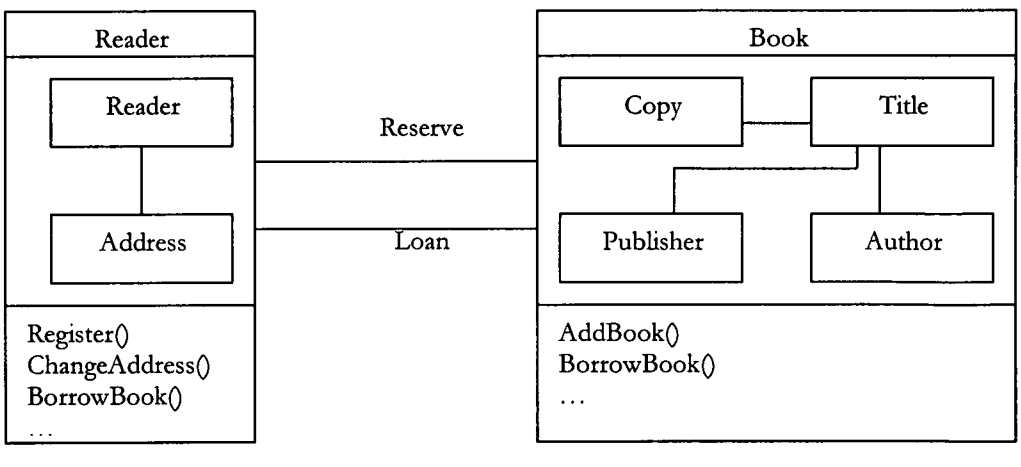
The discussion so far points to a possible solution involving 'classes', based not on individual entities, but larger structures, possibly containing many entities. The emphasis is on finding entities with tight relationships representing large objects with sets of complex behaviour. These 'classes' must therefore take into account both structural as well as functional considerations, and avoid overstretching operations over entities.

Figure 6.6 Library System Entity/Class Diagram



In Figure 6.6, even though there are many entities, NAVITA suggests that there are only two 'classes' in this diagram, as shown in Figure 6.7.

Figure 6.7 Component Composition



These classes can better handle both structural and process changes. Altering the structure such as adding a new reader type, or combining Reader and Address, would only affect the Reader 'class.' Effects of changing some operations can be contained: for example, adding an operation for Book Search will affect only the

Book 'class', although it affects many entities. Some operations such as the one for reservation cancellation, may affect more than one class, yet its effects are more containable. In this way, components become behaviourally richer with a more coherent lifecycle of states. These 'classes' are the basis for NAVITA components.

6.5 Component according to NAVITA

In CBSD methods like this, the concept of component is the central pillar of the method. Since the term component is used in many different contexts – both technical and non-technical – it is impossible to give an authoritative definition of the multi-faceted nature of the term. One can ask numerous questions about the nature of the concept: Are components binary units? Are they executable? Are they non-binary executable units? How are they different from classes and objects? What are components made up of, in terms of interface, implementation, service and package? What is the architectural nature of a component? How are components implemented? How can components be related to business processes? Undoubtedly there are more questions that can be asked. Instead of attempting to address all possible concerns, with the danger of getting distracted from the main issues, our interest is better served by explaining ways in which the term is used in this method and highlighting the key aspects of "component."

6.5.1 Important Aspects of Component

In non-technical terms, components are building blocks of software applications. In a component-based software system, a component is a unit of software that has an interface and an implementation. A good component provides a set of logically cohesive services and has limited dependency on other components. From a more technical point of view, NAVITA software components have the following characteristics:

- A NAVITA component is a package of executable code that encapsulates a well-defined data structure and provides a set of related services through its agreed interface
- Each NAVITA component has a container that enfolds the objects if the component is implemented using an OO technology, or entity instances if the component is implemented using non-OO technology.

Reusing a component involves reusing the container and the structure of classes and entities only, excluding objects and entity instances

- NAVITA business components can be plugged into the Backbone component to share their services

6.5.1.1 Architectural Perspective

As discussed in previous sections, a system can be decomposed along the line of data entities, and processes. As far as component-based decomposition of a system is concerned, NAVITA takes the position, described in Section 6.4.2.7, that components are identified at the point where both data and process decomposition merge. Components need to be neat encapsulations of a mixture of data, processes and various constraints, providing a true unification of data and processes. These components convey a sense of business purpose.

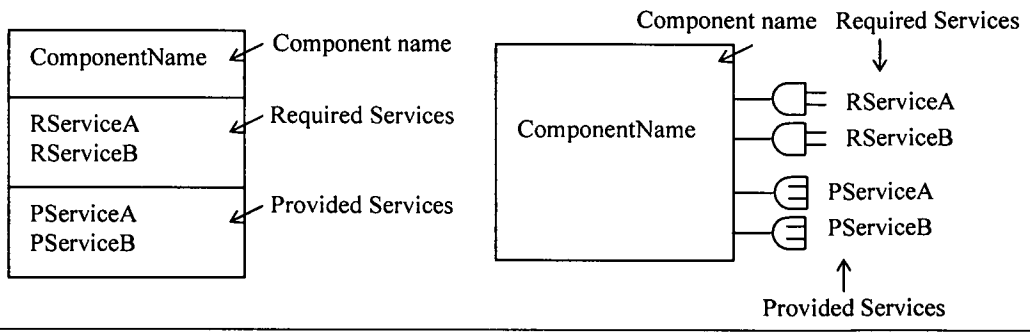
In structural terms, a component has an interface and an implementation of the interface. This separation of interface from the implementation is clearly not a new concept; it can be found in previous generations of methods such as OO and Structured. Yet, the concept has become essential in CBSD methods. Generally each component in an application has a published interface that defines the services provided and required by the component and the implementation of the interface. It is through these interfaces that components define themselves. Components cannot do what has not been defined by their interfaces or contravene them. In some advanced cases, an interface may have many different implementations in order to allow components to behave differently in different situations.

6.5.1.2 Service Perspective

Logical architecture defines the services of the application; how services are translated into components in the physical architecture is determined by the system designer, the method the designer uses, and the availability of exiting components. It is possible to create one business component per service or one business component per many services. Often, using a business component per service would be unrealistic in terms of managing the number of components and their dependencies. Therefore services need to be grouped together in such a way that a component provides related services. In these cases, components are defined in terms of services

they provide. Each component has a published interface that defines the services provided and required by the component and implementation of the interface.

Figure 6.8 Characteristics of a component



6.5.1.3 OO Perspective

From an OO perspective, components are partly similar to and partly different from classes.

Similarities between components and classes

Like classes, components are 'types' and not instances. Both classes and components have namespaces, separate interfaces from implementations (abstraction) and espouse a high level of information hiding. Runtime materialisation is an instance of the component and each instance will have a uniquely identifiable identity. Some components may have multiple instances in an application, while others only a single instance. Therefore, some components may have permanent states. Szyperski (1998) suggests otherwise, but it is not difficult to see the need for components having permanent states. Consider a customer component: if there is a constraint that the total outstanding credit of all customers must not exceed a certain amount, it the customer component has to have a permanent state.

Differences between components and classes

One of the main differences between a component and a class is that a component is has a container, which when instantiated, may contain a number of objects in them. Another important difference is that components provide services and classes operations – activities with lower level of granularity. Certain OO methods advocate the use of responsibility, something bigger than an operation. However, classes do

not have to have these responsibilities; they provide operations not responsibilities, at least not thought of as fundamental property of classes. Components, on the other hand, must provide services. Also, responsibility is a role-playing concept confined to single classes, and services generally encompass more than one class. Components are generally larger than classes, in the sense that they may contain many classes. Therefore, components also act as a grouping mechanism. Components do not expose their own properties, unlike class attributes.

For example, consider Customer as a class and a component. Customer class may have attributes and operations such as to set the credit limit. Customer component may not contain entities/classes, and must provide complete services. For example, it may provide a service to increase the credit limit of everyone by a certain percentage. This would involve retrieving the current credit limit of every object, and then calculating the new limits before the credit limits are updated. This is something an individual customer class cannot do (see the second principle in Section 13.8.1, which is further discussed in Section 4.1.2 of Appendix III).

6.5.1.4 Paradigm Perspective

The concept of component is also paradigm-neutral in the sense that it does not imply the need for implementation in OO languages, although this mostly happens. As an example, the Customer component could, in principle, be implemented as a Relational Database table with the services as queries. Component developers then have the liberty of choosing a specific paradigm. Due to popularity of OO programming languages and support for component-based applications, OO languages are assumed to be the default choice for implementation.

6.5.2 Physical Component Communication

In this architectural model components communicate via the Backbone component. However, during the logical analysis, it has been assumed that components interact directly. The Backbone component is later introduced to this analysis and the model is enhanced in the design stage where implementation issues such as the physical locations of components and the choice implementation languages need to be considered. The primary role of the communication model is to show the rules for

dynamic interactions between components. These rules need to be observed by the components.

6.6 Related Work

Much research has been carried out in the area of software architecture and related issues. Bahsoon and Emmerich (2003) survey architecture evaluation approaches. Researchers at Carnegie-Mellon Institute have made a number of contributions with their publications on a wide range of topics including, evaluation methods (Kazman et al, 1998)) and evaluation of architectures (Gallagher, 2000; Barbacci et al, 1997), design method (Bachmann et al, 2000), and architectural connections (Allen and Garlan, 1994). Shaw has also made a significant contribution to the study of software architecture (Shaw and Garlan, 1996). There has also been much development in the area of architectural patterns (Coplien, 1997; Monroe et al, 1997). Lüer and Rosenblum (2001) describe Wren, a development environment for component-based software.

Chapter Seven

NAVITA System Development Process

7.1 Introduction

A valuable lesson that can be learnt from the rise in popularity of the DSDM framework over more tradition SDPs is that, instead of being either too generic – such as the classical waterfall model – or too specific – such as a process pattern – SDPs should deal with development situations with a set of well-defined characteristics (Stapleton, 1997; Sommerville, 2004; Ambler 1998 and 1999). Accordingly, NAVITA System Development Process (SDP) should be applied only to development of applications with certain characteristics. To this end, NAVITA recognises that there are three general scenarios in CBSD projects. They are:

- An application that exactly or closely matches the users' requirements is found; therefore, the entire application need not be developed
- Some reusable components are found; other components need to be developed and then assembled
- No reusable components are found; components for the entire application need to be developed and assembled

More specific scenarios within these categories are discussed in Section 7.3.

7.2 NAVITA SDP

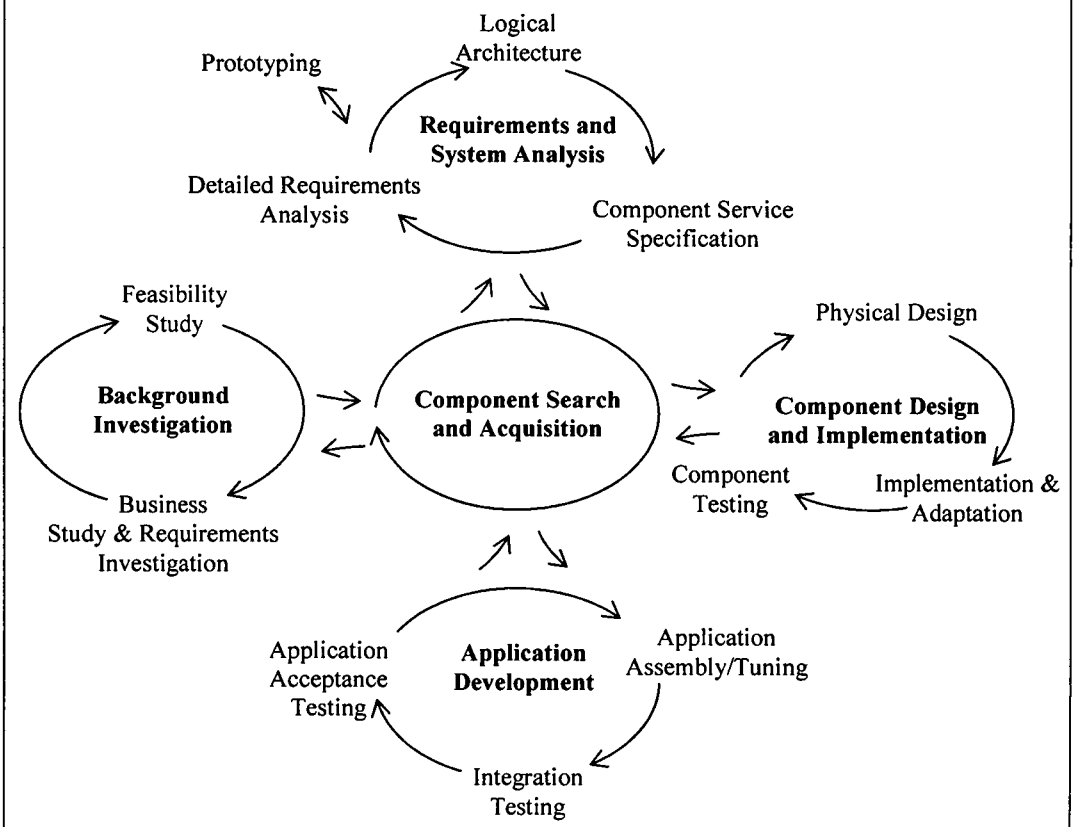
NAVITA SDP is an iterative and incremental process with five main sub-processes. They are:

- Background Investigation
- Requirements and System Analysis
- Component Design and Development
- Application Development

- Component Search and Acquisition

As shown in Figure 7.1, NAVITA puts the search and acquisition of existing components at the heart of the development process. All other development activities are based around this central sub-process.

Figure 7.1 NAVITA SDP



7.2.1 Stage 1 – Feasibility Study

In many ways, this stage is similar to the classic feasibility study; it seeks to establish the viability of the whole project in terms of cost, technology and time. There is a wealth of material discussing issues arising from this exercise, such as (Sommerville, 2004), which need not be repeated here. However, there are CBSD-specific feasibility concerns that need to be considered when the investigation is carried out. They include:

- Availability of Applications/Components – Feasibility reports for CBSD projects need to cover availability of relevant applications and/or components that can potentially be reused in the project.

- Quality, costs and benefits of reusing component – The report should also attempt to outline how software quality, development costs and benefits are affected by reusing existing components (DSDM Consortium, 2000).
- Legal issues – The report should clarify the legal positions of both component supplier and component user regarding ownership and responsibilities.
- Technological issues – Possible barriers to integration and solutions should be highlighted in the report.

Before or while establishing the feasibility of a project, the business context and the users' requirements need to be investigated. The width and depth of this business and requirements analysis inevitably depends upon the size and complexity of the application, as well as the analysts' familiarity with the business domain. Nevertheless, it is clear that Stage 1 and Stage 2 intertwine.

7.2.2 Stage 2 – Business Study and Requirements Investigation

In this stage, the analysts attempt to establish a good understanding of the business domain and the users' requirements through analyses of the business processes and the requirements for a new system. During or immediately after the analysis of business processes through Business Process Modelling (BPM), the analysts attempt to capture the user's requirements by means of requirements elicitation techniques (Kotonya and Sommerville, 1998; Sommerville, 2004). Although the requirements investigation is often preceded by the BPM, the two analyses are done in a cyclic way. BPM also involves envisaging how the current business processes may be affected by the new system. Therefore, business process models are often produced for the current and desired states of the business processes.

NAVITA recommends using the BPM techniques discussed by Allen and Frost (1998) in conjunction with the standard Business Process Modelling Language (White, n. d.; BPMI, 2004). For the analysis of the users' requirements, NAVITA provides the Context Diagram (Chapter 8), which is mandatory for most development situations. If the requirements need to be captured in greater detail, the Middle-level Functionality Diagram (Chapter 9) can be deployed.

7.2.3 Stage 3 – Component Search and Acquisition

This is the one of the most essential activities of CBSD. So far in the development, analysts would have a) established the feasibility of the project b) understood the business context and c) found out the functional and non-functional requirements of the system. Equipped with this knowledge, developers can first look for an application that may satisfy the requirements. Candidate applications are verified and validated by developers and users. In early iterations, this need not be a rigid one-way process: often users' requirements can be renegotiated if similar application(s) are found (Sommerville, 2004). The development can loop back to the **BACKGROUND INVESTIGATION** sub-process a few times.

There are two possible outcomes from this initial search. The first scenario is that a matching application – perhaps after some renegotiation with the users – is found and may be adapted, before being verified and accepted by the users. The second scenario is that no matching application is available or its reuse unviable. In that case, the system is further analysed and broken down into components, in terms of their specification, in the **REQUIREMENTS AND SYSTEM ANALYSIS** sub-process. After that, developers then look for components satisfying these specifications. Only those components that are not available or viable for reuse are developed in the **COMPONENT DESIGN AND DEVELOPMENT** sub-processes. Once all the necessary components are acquired, the application is assembled in the **APPLICATION DEVELOPMENT** sub-process.

In a broader sense, the **COMPONENT SEARCH AND ACQUISITION** stage involves the following activities:

- provide mechanisms for storing, searching and retrieving existing applications and components from a range of sources including internal repositories
- verify applications and components against their specifications
- maintain portfolios and/or libraries of reusable applications and components
- provide legal and technical expertise necessary for the acquisition of internal and external components, possibly from independent component vendors.

Much research has been done in this area of CBSD. For example, Berglund (2002) discusses issues related to documentation of component libraries while Zaremski and Wing (1997) show how formal specifications can be used for specification matching. Yao and Etzkorn (2004) examine various approaches to component classification and retrieval before describing how semantic-web can be used for component classification and retrieval.

7.2.4 Stage 4 – Detailed Requirements Analysis

Once it is clear that reusable applications are either unavailable or unviable, analysts have to decompose the system in greater detail to uncover the logical components of the system. Since the initial investigation into users' requirements in the **BUSINESS STUDY AND REQUIREMENTS INVESTIGATION** sub-process is not detailed enough, before the components can be identified, analysts perform more detailed investigations into the users' requirements. This investigation is supported by prototyping, usually using pen and paper. For iterative development, the **BUSINESS STUDY AND REQUIREMENTS INVESTIGATION** sub-process can be repeated.

The key NAVITA models used in this stage are Middle-level Functionality Model (Chapter 9), Lower-level Functionality Model (Chapter 9) and Information Model (Chapter 11).

7.2.5 Stage 5 – Prototyping

Various prototyping techniques can be deployed depending on the development scenario. For instance, a paper-based mock-up of the user interface (or another form of throw-away prototype) of the new system can be used when it is likely that the component will be acquired from a vendor. On the other hand, if it is reasonable to believe that the component will need to be developed (internally), then a set of reusable interfaces could be developed. The primary aim of prototyping is to help capture important user requirements.

NAVITA provides Logical Screen Layout (Chapter 10) and User-System Dialogue Model (Chapter 10) for this development stage.

7.2.6 Stage 6 – Logical Architectural Analysis

Once the analysts have gained a good understanding of what the system should do, they are in a position to break down the system into smaller components and specify them. This is a high-level analysis of the application – or part of an application, if an incremental approach is used – in order to understand the logical composition of the system and services of the components.

NAVITA Protocol Model (Chapter 12) is used to analyse the interaction between logical components in this stage.

7.2.7 Stage 7 – Component Service Specification

Following up on the previous stage, analysts bring together various models to produce precise specifications services of components. Production of the service specifications will lead to a renewed search for components satisfying these services as defined in the specifications. There are two main possible outcomes to this search: the first outcome is that components are found which provide all the required services and these components need not be modified in any form, and the second outcome is that all, some or no components are found, and in cases of component(s) being found, at least some of them will need tuning. The first outcome will lead to the **APPLICATION DEVELOPMENT** sub-process and second, the **COMPONENT DESIGN AND DEVELOPMENT** sub-process.

Models required for service specifications are described in (Chapter 12).

7.2.8 Stage 8 – Physical Design

Once it is established that a component providing particular service(s) cannot be reused, a new component needs to be designed and implemented. In cases where no reusable component of any form exists, physical design of components may be produced from scratch. This involves precisely specifying how the components should be implemented using particular technology. In cases where an adaptable component exists, its specification may be used as a starting point for re-design. Such technology-dependent designs for physical components are produced at this stage.

NAVITA provides detailed guidelines on production of designs for reusable components in Chapter 13.

7.2.9 Stage 9 – Implementation/Adapt

If an existing component cannot be found to the specification of a service, then the component needs to be developed anew. If an adaptable component exists, it is modified accordingly. Much research is done on issues related to component adaptation concepts and technologies. Davis (1995), Jacobson et al (1997) and Atkinson et al (2002) discuss various component adaptation mechanisms, whilst Campbell (1999) and Heineman (1998) suggest how implemented components can be adapted.

7.2.10 Stage 10 – Component Testing

Developed components are then tested to see if they do what is intended. This is part of what is known as unit testing in software engineering (Sommerville, 2004). A lot of research has been done in this area; for example, see (Wittenberg, 2000; Bertolino and Polini, 2003; Ramachandran, 2003; Brinkmeyer, 2005)

7.2.11 Stage 11 – Application Assembly/Tuning

If a matching application is found, this step is skipped. If the application needs tuning (tuning means refining certain aspects of a component/application which does not require changes to the design), it is done to the application and development proceeds. If components are either found or developed, they are assembled to create the application.

7.2.12 Stage 12 – Integration Testing

This is the testing of whether or not components and/or applications communicate with each other in the predetermined way without deviation.

7.2.13 Stage 13 – Application Acceptance Testing

Alpha and beta tests are carried out at this stage before the final delivery of the system.

7.3 NAVITA SDP Scenarios

In terms of the nature of development, NAVITA is applicable to CBSD projects where there is a good opportunity for reusing existing components. This method covers the following project scenarios.

Table 7-1 CBSD Development Scenarios and NAVITA SDP Routes

Development Scenarios	NAVITA SDP 'Routes'
Matching application found: No development or modification required	Background Investigation Component Search and Acquisition
Similar application found: Modification/Tuning required	Background Investigation Component Search and Acquisition Application Development
All components found: Needs assembling	Background Investigation Component Search and Acquisition Requirements And System Analysis Application Development
Some components found: Some needs to be developed, then assembled	Background Investigation Component Search and Acquisition Requirements And System Analysis Component Design And Development Application Development
Some/All components need modification: Adapt components and assemble	Background Investigation Component Search and Acquisition Requirements And System Analysis Component Design And Development Application Development
No components found: All components need to be developed, then assembled	Background Investigation Component Search and Acquisition Requirements And System Analysis Component Design And Development Application Development
Add/Remove components from application: Analyse implications and perhaps with some tuning too	Background Investigation Component Search and Acquisition Requirements And System Analysis Component Design And Development Application Development
Upgrade existing components: Redesign and adapt components	Background Investigation Component Search and Acquisition Requirements And System Analysis Component Design And Development Application Development

7.4 Conclusion

By placing the COMPONENT SEARCH AND ACQUISITION sub-process at the centre of the SDP, NAVITA makes clear that reuse of the entire application is the top priority after the initial investigation is carried out in the BACKGROUND INVESTIGATION sub-process. Only when such a reuse is not possible – even after renegotiations of users' requirements – should the system be analysed in greater detail. Not all CBSD methods emphasise this important point in their SDPs. When components of a

system are identified, SDPs of CBSD methods need to make clear distinctions between the activities related to the identification specification of components from the activities related to the design and development of the components. This distinction should be reflected in the modelling techniques too. NAVITA SDP reinforces this distinction by dealing with these two sets of activities separately in the **REQUIREMENTS AND SYSTEM ANALYSIS** and **COMPONENT DESIGN AND DEVELOPMENT** sub-processes respectively. Components may be designed and developed from scratch or from existing components in the **COMPONENT DESIGN AND DEVELOPMENT** sub-process, before they are assembled in the **APPLICATION DEVELOPMENT** sub-process. These sub-processes can be executed iteratively and incrementally.

Chapter Eight

NAVITA System Modelling
Context Diagram

8.1 Introduction

The Context Diagram has been used by a number of system development methods, Structured and OO alike, to establish the system's boundary, its environment and the interactions between them. This chapter presents a critical examination of the Context Diagram as traditionally understood, highlighting its important limitations. The chapter then discusses how these limitations are tackled in this method by presenting the concepts, modelling process and technique used in the NAVITA Context Diagram, and exploring how the development of this diagram fits into the overall system development process and software architecture.

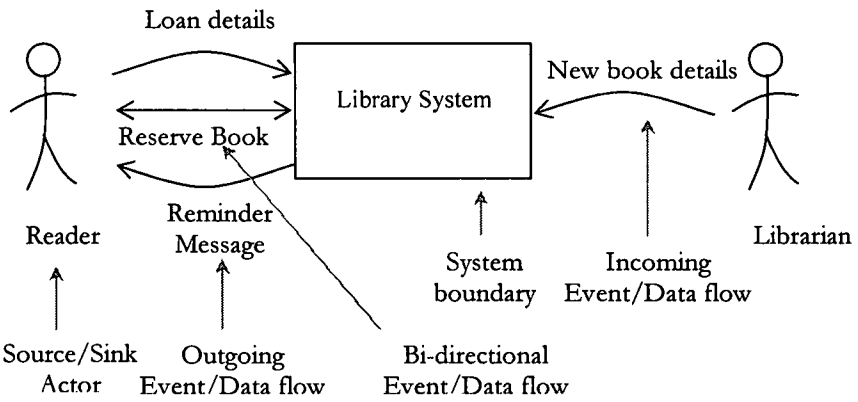
The Context Diagram is the first system model developed in NAVITA, and it is used in the investigation and definition of users' requirements. Development of a context diagram enables analysts to explore:

- the roles of the users of the new system
- the way users will interact with the system and amongst themselves
- the outline of the new system's functionality
- the manual and other procedural aspects of the new system and possible interface(s) with other system(s).

8.2 Context Diagram

Before proposing the NAVITA Context Diagram, it is worth discussing the way the diagram is used in other methods and limitations of its use. Traditionally, context diagrams are used to define the system's boundary by means of analysing the interactions between the system and the entities in its environment, such as external entities or sources/sinks in Structured methods and actors in OO methods. Figure 8.1 shows a simple context diagram using a mixture of symbols and concepts from different methods, ones which both Structured as well as OO analysts will recognise.

Figure 8.1 Context Diagram as Traditionally Understood



8.2.1 Limitations of Context Diagram as Traditionally Understood

There are two major assumptions in the traditional way of thinking about context diagrams.

First, there is an assumption that those who are involved in this analysis would very quickly, if they had not already, come to conclusions about what is inside and what is outside the system's boundary; for example, see (Yourdon, 1989). In other words, in one of the very first diagrams developed for requirements analysis, the system's boundary will be established, albeit in somewhat abstract terms. It has been well documented that because users are often unsure about what can or should be done by the new system, defining the system's boundary is usually a difficult and lengthy exercise (Kotonya and Sommerville, 1998; Loucopoulos and Karakostas 1995; Vliet, 1993). Realising the volatility of the requirements, and the system's boundary by implication, DSDM for example, has proposed a feature called MoScoW rules, which is used to prioritise the requirements (Stapleton, 1997). As far as modelling is

concerned, there is a need for a diagram that can help analysts explore the grey area of the system's boundary and make a conscious decision about what the system will and will not do.

Second, the traditional approach to developing context diagrams also assumes that the system exclusively means the 'computerised system'. As pointed out by Jayaratna (1994), this notion of information system as a system of computerised processes is unnecessarily restrictive. In fact, it is useful to think of IS as a computerised system combined with the supporting manual aspects. For example, a librarian may refuse to register a reader with the system if the reader has not provided certain documentary evidence. Checking such documents may not necessarily be part of the computerised system; nevertheless it is an essential part of the registration process and one that analysts need to take into account when analysing the information system. Information systems therefore not only include computerised processes of the system, but also manual procedures and human decisions too. Often, there are complex and changing interactions between computerised and manual processes and thorough investigation of the system functionality requires careful analysis of these interactions.

8.2.2 The NAVITA Context Diagram

The NAVITA Context Diagram is in some ways similar to the diagrams used in popular Structured and OO methods, in the sense that it is also used to analyse the system's environment and interactions with the system. In addition, the NAVITA Context Diagram also attempts to address the two issues discussed in Section 8.2.1. This means that the context diagram will emphasise that the definition of the system boundary involves careful consideration and decisions about what falls and does not fall within the system boundary and the importance of non-computerised aspects of the system. The NAVITA context diagram allows analysts to explore these complex issues.

8.3 Context Diagram: Modelling Concepts

The main concepts used in this modelling are:

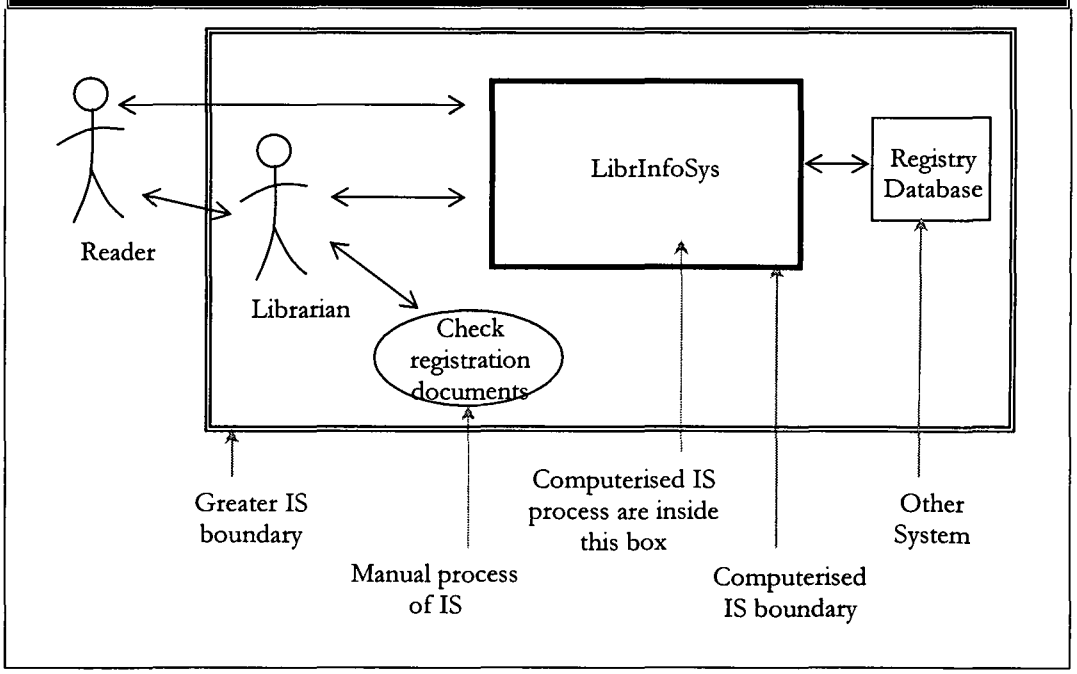
- System Boundaries
- Actor

- Interaction

8.3.1 System Boundaries

The system boundary, represented by a rectangular box, is used to denote the system under investigation, as well as other systems it interacts with. What is inside the boundary of the system being analysed is traditionally understood as an information system, and what is outside, the business environment. The boundary effectively boxes in the functionality of the information system to be designed and implemented, as shown in Figure 8.1. NAVITA however suggests another kind of system boundary, named the 'greater information system boundary', which shows the system as a whole, inclusive of both the computerised IS as well as manual processes that are essential to the running of the computerised IS. Since the boundary of the computerised information system is not necessarily identical to the boundary of the 'greater' information system, the two boundaries are shown separately. From this point of view, the boundaries of an information system and a computerised information system can be depicted as shown in Figure 8.2.

Figure 8.2 NAVITA Context Diagram



In the diagram, computerised activities are boxed inside the IS boundary, represented by the single-lined rectangle, while the greater IS boundary is represented by a double-lined rectangle. The two boundaries sandwich manual activities, some actors and other system(s). The manual activity Check registration

documents is inside an oval, like a computerised process. The actor Librarian between the two boundaries is regarded as being part of the greater IS because it is perceived as such by its business actors, Reader. Librarian acts as the human interface to the computerised IS. Other external systems that the computerised IS interacts with are represented by single-lined rectangles, such as the Registry Database. Outside the greater IS boundary, there are actors transacting with the business.

8.3.2 Actor

Actors are represented by stickman figures, as in UML. Jacobson et al (1992) explains the concept as follows:

> Actors model the prospective users; the actor is a user type or category, and when a user does something he or she acts as an occurrence of this type. One person can instantiate (play the roles of) several different actors. Actors thus define roles that users can play.

UML v1.5 (OMG, 2003) gives the following definition of actor.

> An actor defines a coherent set of roles that users of an entity can play when interacting with the entity. An actor may be considered to play a separate role with regard to each use case with which it communicates.

These definitions are vague about the responsibilities of actors in relation to the system.

8.3.2.1 Actor Responsibilities

NAVITA recognises that, from the perspective of system functionality, actors have three important responsibilities. These are to:

(a) Generate (Business) Triggers

Actors generate the business trigger or event leading to the execution of certain system functionality. Actors may generate triggers by sending data, time signals or control to the system. Actors in this role are called First Actors (Allen and Frost, 1998). These actors also serve as a link between BPM and system modelling.

(b) Interact Directly with the System

Actors use or directly interact with the computerised system by exchanging information through some kind of dialogue, either online, offline or a combination of both. Actors in this role are called Operator Actors.

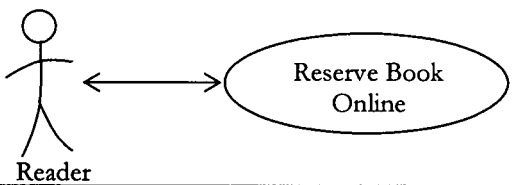
(c) Be Served by the System

Actors, mainly representing people for whose requirements the system functionality aims to satisfy, are duly served by the system. These actors are called Final Actors.

These three roles are present in the use of every system functionality. In some cases, the same person or entity can be involved in all of these three roles and it is not necessary to distinguish roles. In other cases, where different actors play different roles, it becomes necessary to distinguish the roles of actors. This can be illustrated by the following three examples.

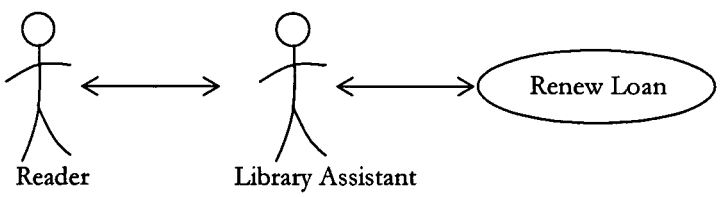
The first example can be regarded as a simple use case diagram showing a system functionality Reserve Book Online, which is used by the actor Reader.

Figure 8.3 Reader in First Actor, Operator Actor and Final Actor Roles



Reader is the first actor because it is his or her sending of details to reserve a book which generates the business trigger. Reader is the operator actor because he or she interacts directly with the system and Reader is also the final actor because the system is designed to serve such an actor. Therefore, having the actor reader linked with the reserve book functionality is sufficient. This is the notion of actor envisaged by OOSE and UML.

Figure 8.4 Reader as First and Last Actor, Library Assistant as Operator Actor

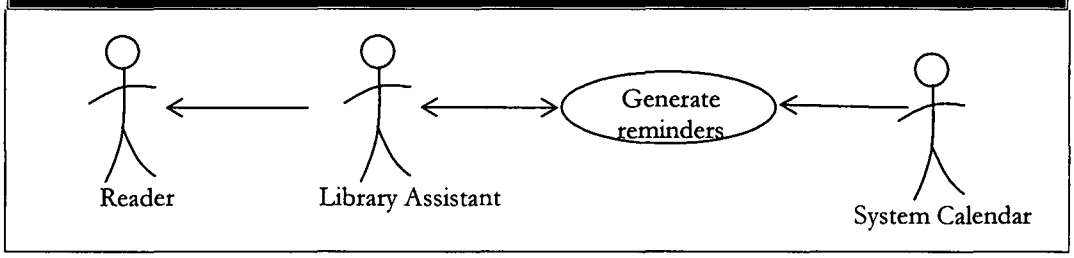


In the second example, the functionality is Renew Loan, designed for readers who renew their loans by telephoning the library assistant who then renews the loans on the system.

In this case, Reader is the first actor because it is his or her sending of renewal details that generates the business trigger, Library Assistant is the operator actor because he or she interacts directly with the system and Reader is the final actor because it is for dealing with their needs that the process is carried out.

The third and final example is the situation in which the library system automatically generates and prints a list of reminders that are collected and sent by library assistants to readers who did not return their loans on time.

Figure 8.5 Reader as Last Actor, Library Assistant as Operator Actor and System as First Actor



System Calendar or perhaps another active part of the system that generates the trigger is the first actor, Library Assistant is the operator actor and Reader, representing those who have not returned their loans on time, is the final actor because it is to remind them of overdue loans that the functionality is designed.

8.3.2.2 Actor Types

Actors can be categorised on different basis (Allen and Frost, 1998; Armour and Miller, 2001). In terms of their placement in the system boundaries, actors can be divided into three categories: System Actor, Operator/Internal Actor and Business Actor.

System Actor

System Actor represents an active element of a system which is capable of firing off a trigger to which the system needs to respond. A successful system response to such a trigger would constitute a piece of system functionality that is of interest to other actors. System actors are often non-

human entities, such as internal clocks, active objects (Gomaa, 2000), signalling devices and subsystems. See for example, the System Calendar actor in Figure 8.5.

Internal or Operator Actor

Operator Actor represents an entity that is considered as external to the computerised IS, but internal to, or part of, the greater IS. Human internal actors work as interfaces between business actors and the computerised IS by facilitating the interaction between the Business Actors and the system, such as Librarian in Figure 8.6. In cases of lesser computerisation of business processes, these actors may carry out part of the activities and decision-making; for example, a library system in which librarian, rather than the computerised IS, decides if a book is for short or long term loan. In more computerised systems, such decisions may be made by the computerised IS. In the case of total computerisation, these actors will cease to be part of the system. There will only be direct interactions between business actors and the computerised IS; for example, e-commerce applications (Ince, 2003).

Business Actors

Business Actors are external to the greater IS, representing business entities that the IS serves or transacts with. These actors may be either internal or external to the business organisation. Customer and Supplier are classic business actors that are external to the business. Employee, in the case of a payroll system, and Lecturer, in the case of university ISs, are classic business actors that are internal to the business, yet external to respective ISs. In Figure 8.6, Reader and Book Supplier are business actors.

8.3.3 Correlations between Actor Responsibilities and Actor Types

The correlations between actor responsibilities and actor types discussed in the previous two sections are shown in Table 8-1.

Table 8-1 Correlations between Actor Responsibilities and Actor Types

Actor Responsibilities	Actor Types: Business Actor	Operator Actor	System Actor
Generate trigger	Yes	Yes	Yes
Use the system	Yes	Yes	No
System aims to serve them	Yes	No*	No

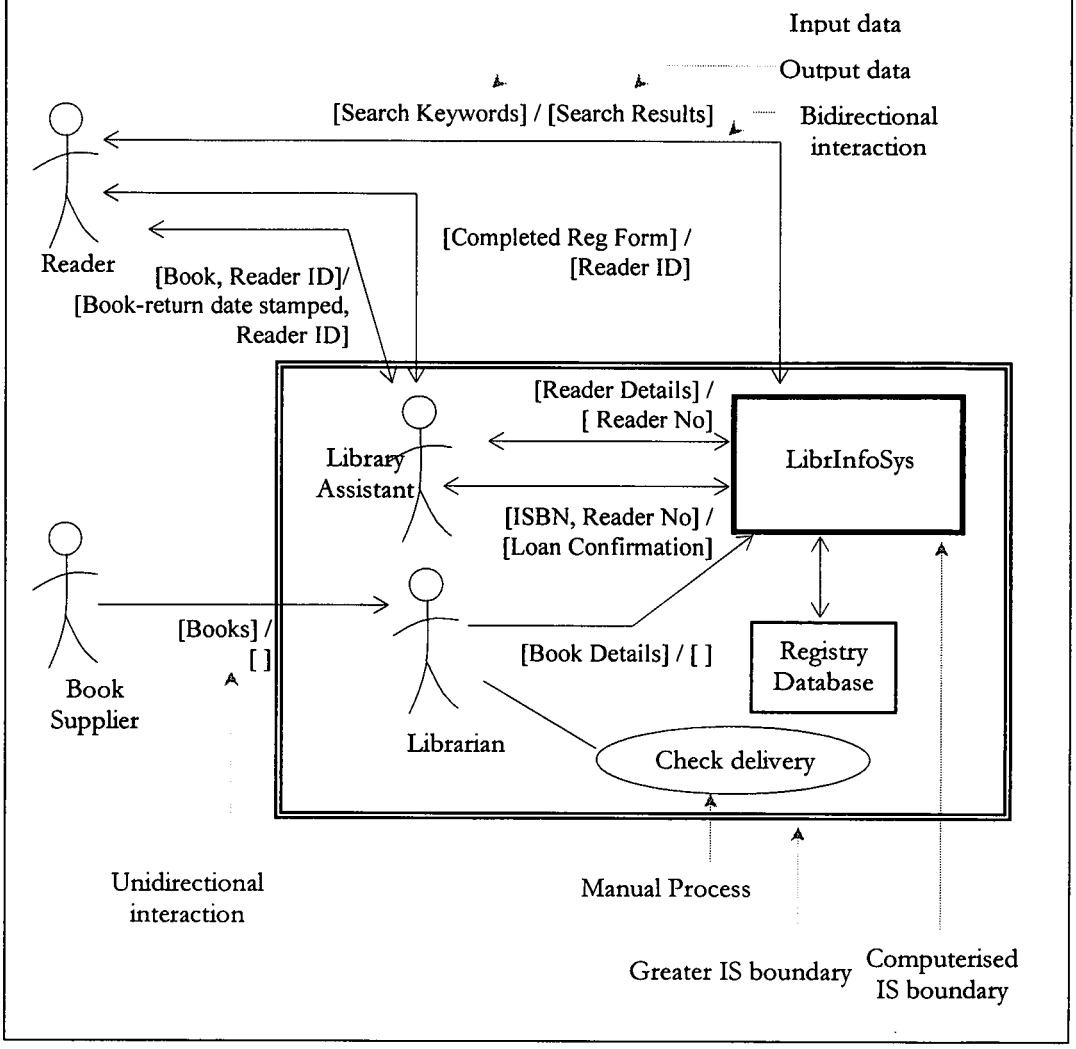
* Not directly, although it can be argued that they also benefit indirectly by using the system.

8.3.4 Interaction

Interactions denote the communications between actors, actors and system, systems, and actors and manual processes, as shown in Figure 8.6. Interactions can be either unidirectional or bidirectional. Interactions between actor and the system are always logical because they carry information only. However, interactions between actors can be physical and/or logical. When naming interactions, the following rules are applied:

1. arrowheads are appropriately used to indicate the directions of information flow – single headed from source to target if unidirectional, double headed if bidirectional
2. interactions are named either from left to right or top down
3. "[a] / [b]" denotes a bidirectional interaction, "a" is input information and "b" is output information
4. if the interaction is between two actors, "a" and "b" can be either logical information or physical things
5. "[a] / []" and "[] / [a]" denote input only and output only unidirectional interactions

Figure 8.6 Partial Context Diagram for LibrInfoSys



8.4 Context Diagram: Modelling Process and Technique

Context diagrams are usually developed with user involvement and the process is often iterative.

CD Step – 1 Represent the system under investigation using a single-lined rectangle, with the name of the system inside it. Draw a larger double-lined rectangle to denote the greater information system. Internal actors may be identified in many cases; if there are none, this outer boundary will be removed later.

In the example in Figure 8.6, the system is represented inside a box and named as LibrInfoSys, and a larger double-lined

rectangle is also drawn for the greater information system.

CD Step – 2 Identify all potential users and those who will interact with the system, with the help of various stakeholders from the business domain through appropriate application of requirements elicitation techniques such as interviewing, JAD (Wood and Silver, 1995). Determine the business roles of those who will be using the system. Confirm the actors and their roles with project stakeholders.

In the example, the following actors are identified by envisaging the potential users of the system: Reader, Library Assistant, Librarian and Book Supplier.

CD Step – 3 Analyse the type of each potential actor and plot its placement in the boundaries accordingly.

In this case, Librarian and Library Assistant are operator actors because they function as part of the IS by facilitating the other actors interactions with the system, and Reader and Book Supplier are business actors because they are the entities the business transacts with. For further explanations, see Section 8.3.2.2.

CD Step – 4 Identify external systems that the main system may need to interact with by examining whether the system is part of the large set of applications, whether any of the data it holds is shared by other systems and whether it depends on the input from other systems.

In the example, the Registry Database is regarded as the external system because it holds detailed information about students who are potential readers of the library.

CD Step – 5 Identify the interactions between the various elements identified so far by looking at flows of information and material goods. Prototypes of user interfaces, reports and documents are used to validate the interactions. It should be emphasised that when

identifying these interactions, and in particular those between the actors and the system, each interaction should be a neat encapsulation of detailed exchanges between the actor and the system to carry out a complete business process. These interactions will give analysts a sense of what the system will have to do in dealing with these inputs and outputs.

In the example, [Search Keywords] / [Search Results] neatly sums up the interactions required between the actor and the system in order to complete a catalogue search. On closer inspection, analysts may discover finer details but these are not important. At this level of abstraction interactions should reflect the system's functionality such as a catalogue search facility.

CD Step – 6 Based on the interactions and initial understanding of the system functionality, identify manual aspects of the system, and other systems it may interact with. If no internal actors, manual activities and external systems are relevant to this system, the greater system boundary should be removed.

In the figure, Book Supplier delivers material books to the Librarian who may check these books against the order and delivery notes and examine the books' physical condition, before entering the information into the system. In this case, the physical activity is included because, the assumed business rule is that the details of the book are not to be entered if the above three criteria are not met. Therefore, it is an important part of the process of entering new book information to the system, hence, included in the diagram.

CD Step – 7 After discussions with the project stakeholders on the boundaries of the system, remove actors and interactions that are outside the concern of the project. For example, the external actor Book Supplier can be removed if the library system does not need to know anything about the suppliers of the books. In these cases,

the associated interaction with the Librarian and the manual process Check Delivery should also be removed.

CD Step – 8 Consolidate the actors by ensuring that all actors have distinct roles. For example, Library Assistant and Librarian should have a unique set of responsibilities, from the perspective of the IS, to be regarded as separate actors. In this case, the assumption is that only Librarians can maintain the book catalogue and the Library Assistants are for providing reader services. If both actors have an identical set of roles, they have to be combined, and possibly given appropriate names.

CD Step – 9 Consolidate interactions by removing unnecessary input and output data items from the interactions. For example, it may not be necessary to input all reader details to register a loan with the system; reader ID may be sufficient.

CD Step – 10 Document the diagram using templates such as those suggested in Section 8.5 and discuss the model with the main stakeholders from the business domain.

It should be emphasised that because the context diagram focuses largely on the system's environment and not its functionality, although the interactions may provide some clues about how the functionality may appear later, what the analysts can glean about the exact nature of interactions, manual processes and other system may be somewhat limited. Only when further analysis of system functionality is carried out, can analysts have better knowledge about these, and so append and revise this diagram accordingly. Therefore, development of context diagram and analysis of system functionality go hand in hand, making the process iterative.

8.5 Context Diagram: Documentation

The context diagram is documented mainly by describing the actors, manual processes, external system(s) if any, and the interactions between these elements. Authors have proposed different templates for documenting actors, use cases, classes etc (Cockburn, 2000; Armour and Miller, 2001). Furthermore, CASE Tools

such as Rational ROSE and Poseidon also provide standardised templates for documenting them (Rational Rose, n. d.; Poseidon, n. d.). Such diversity illustrates that perhaps there can be no one-size-fits-all solution to this issue; the nature and complexity of the system will dictate how much detail these descriptions should contain. It is up to analysts to judge and choose an appropriate documentation standard or create a customised one if necessary. NAVITA suggests the following documentation templates.

8.5.1 Documenting Actors

Documentation of actors includes a description of the actor, and their business roles and responsibilities.

Figure 8.7 A Simple Template for Documenting Actors

Actor No: ---------

Actor Name: --------

Description: ----------

Business Roles & Responsibilities: ----------

8.5.1.1 Documenting Manual Processes

Manual processes can be documented by describing them in a simple step-by-step manner.

Figure 8.8 A Simple Template for Documenting Manual Processes

Manual Process No: ---------

Actor Name(s): --------

Description:

Step 1: ----------

Step 2: ----------

Relationship with Functionality Unit(s): ----------

8.5.1.2 Documenting External Systems

External systems can be documented by describing their main functionality only from the point of the new system; there is no need to describe aspects of the external system's functionality that are not relevant to it. The nature of interactions between the system will indicate what functionality is of interest to the system.

Figure 8.9 A Simple Template for Documenting External Systems

System Name: ---------

Relevant Aspects: ----------

8.5.1.3 Documenting Interactions

Interactions can be documented by detailing the purpose of the interaction, and inputs and outputs.

Figure 8.10 A Simple Template for Documenting Interaction

Interaction No: ---------

Purpose: ----------

Input Data Items: ----------

Output Data Items: ----------

8.6 Development Process

This modelling is used in Stage 1 – Feasibility Study and Stage 2 – Business Study and Requirements Investigation. This is a mandatory model for all scenarios of development. It provides the basis for the rest of system modelling.

8.7 Software Architecture

The interactions between the actors and the system, and the system and the external system described in NAVITA Context Diagram are significant to the software architecture. These interactions form the basis for identifications of the system functionality and its interaction with external actors, which the architectural model must take into account.

Chapter Nine

NAVITA System Modelling
Functionality Model

9.1 Introduction

One of the main aspects of system development is the production of a precise requirements specification. The specification itself may take a variety of forms, from a simple document with a narrative list of various functional and non-functional requirements to a set of highly formalised mathematical expressions. Like popular system development methods, NAVITA suggests using semi-formal diagrams accompanied by detailed documentation. Exactly which diagrams and documentation make up the requirements specification in NAVITA will vary, because of the nature of component-based development. It is clear that the context diagram will be necessary for most scenarios, and in many cases, further analysis will be required. This chapter examines the most popular requirements modelling technique at present, use case modelling, and explores its limitations, before discussing the proposed requirements analysis approach.

The main aims of functionality modelling, or requirements analysis, in NAVITA are as follows.

- To understand what the users want the new system to do
- To ensure consistency and completeness of the requirements through rigorous modelling
- To project the requirements into a set of coherent models

Context Modelling, discussed in Chapter 8, has established the system's environment in terms of its boundaries, the roles of various users and the interactions among the system, its actors, manual processes and other collaborating systems. An abstract outline of the system, therefore, has been put into place. It is now important to define clearly what the system will do from the user's perspective.

Requirements elicitation and modelling is known to be a deceptive exercise because it is a multi-dimensional process involving psychological and political considerations rooted in ambiguities and uncertainties (Stapleton, 1997) as well as technical problems. Such non-technical problems are numerous. For example, users often do not know what they want; how to express what it is that they want; requirements often change due to organisational and learning-curve factors. There are also numerous accounts of situations where the developers finish a product only to find that the users have changed their minds about what the system functionality should be. This does not, however, diminish the need for capturing requirements; rather, system development should be seen as, to some extent, an exploratory, dynamic and iterative process that calls for mechanisms to control changes introduced to the development process. In this sense, the component-based approach seems to help resolve the problem, by enabling developers and users to add, remove and modify the functionality of the system instantaneously.

9.2 Functionality Modelling

There are a number of techniques, from Structured to OO, which help analysts capture the requirements of the users. In the early days of SDMs, diagrams such as flowcharts were used to aid analysis of requirements (Boillot et al, 1995). The Data Flow Diagram was a popular requirement modelling techniques with Structured methods, even making its way into earlier OO methods such as OMT (Rumbaugh et al, 1991; Derr, 1995). Since there were thought to be some inconsistencies between DFD and other OO models, mainly the class model, methodologists attempted to come up with a new OO requirements analysis technique. The publication of the use case modelling in OOSE (Jacobson et al, 1992) has attracted major attention due to its acclaimed simplicity, and the ease with which novice users can learn and understand the diagram. The inclusion of use case model in UML and the amount of

literature produced on the topic is a clear testament of its popularity (Rosenberg and Scott, 1999; Armour and Miller, 2001; Leffingwell and Widrig, 2003).

Therefore, this research takes the use case modelling as a reference point, and will now examine it critically and suggest a novel approach that will overcome the noted deficiencies.

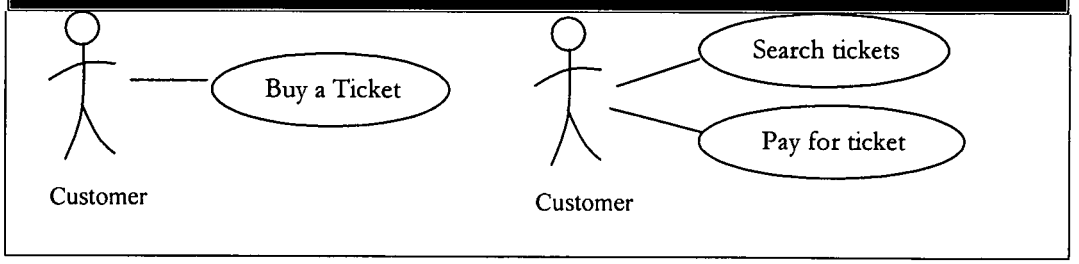
9.2.1 Use Case as Functionality Model

Section 8.3 provides a detailed examination of weaknesses of the two concepts, actor and system boundary, used by the traditional use case modelling approach. Now, attention will be turned to the central concept, the use case itself. When it was first published (Jacobson et al, 1992), use case was defined as follows.

> "A use case is a sequence of transactions in a system whose task is to yield a measurable value to an individual actor of the system."

Perhaps, the most puzzling thing about the definition and application of the use case concept is its granularity. A simple example can expose this; Buy a Ticket will be one use case, and Search Tickets and Pay for Ticket will be two use cases even though in practice the first single use case and the latter two combined are semantically equivalent. See Figure 9.1.

Figure 9-1 Problem with granularity of use cases

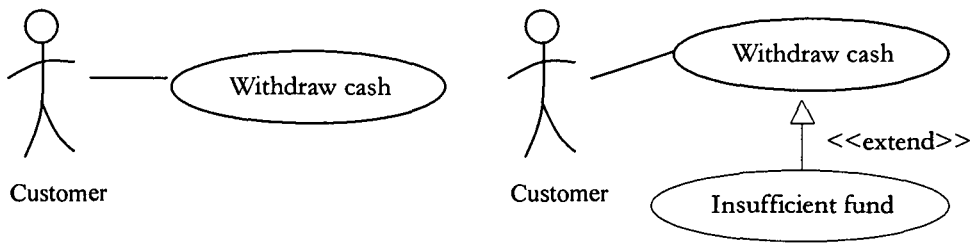


To make the matter worse it seems, the concepts of <<uses>> and <<extends>> are used to show dependencies between use cases[7]. Often, the concept of use case is used to model transaction-like processes and at the time same time these stereotyped relationships allow individual use cases to remain incomplete. For example,

[7] When the use case model is integrated into UML, the relationships are often called <<include>> and <<extend>>. In addition to these, UML also allows other stereotyped relationships between use cases (OMG, 2003).

Withdraw Cash is a use case, so are Withdraw Cash extending Insufficient Fund, as shown in Figure 9.2.

Table 9-2 Complete and incomplete use cases – both are valid

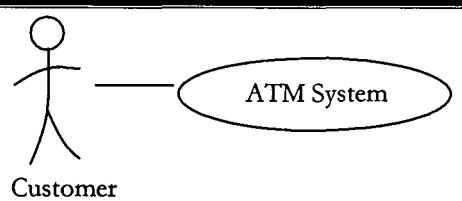


The concept of use case is therefore employed to model both complete and partial sequences of transactions. This strongly indicates that the definition of use case is self-contradictory. At the same time, Jacobson himself has widened the concept to model the business processes with the so-called business use case, which is defined as follows (Jacobson, 1996).

> A Business Use Case defines a sequence of events that provide value to business actors. Business actors are roles fulfilled by individuals, organizations, or systems that exist external to the business.

In this definition, a use case is no longer 'transactions in a system', but something that can also include manual aspects too. In another publication, Jacobson goes even further and suggests that entire application or system can be a use case too, known as a superordinate use case (Jacobson et al, 1997), see Figure 9.3.

Table 9-3 A Superordinate use case



The UML Specification v1.5 provides the following definition of use case (OMG, 2003).

> The use case construct is used to define the behaviour of a system or other semantic entity without revealing the entity's internal

structure. Each use case specifies a sequence of actions, including variants, that the entity can perform, interacting with actors of the entity.

9.2.2 Main limitations of the concept of use case

After a few years of being practised widely, authors and researchers, such as (Glinz, 2000), have made various criticisms, interpretations, reinterpretations and suggestions for improvement for use case modelling. Some of these suggestions often overload rather than clarify the concept. What is clear, however, is that despite its simplicity and flexibility, the concept of use case modelling is perhaps too simplistic for modelling rich system behaviour. In summary, use cases have the following major weaknesses.

- There is an uncertainty about the granularity of use cases.
- There seems to be some contradiction in the definition of use case, particularly in the light of use case relationships concepts such as <<extends>> or <<extend>>, and <<uses>> or <<include>>.
- There is no concept of decomposition of the system in use case modelling. When dealing with system functionality of a highly hierarchical nature, analysts have difficulties expressing such nature. Often, they are inclined to employ use cases to show this and can do so because of the loose granularity of use case. For others, it means more confusion.

9.2.3 NAVITA Functionality Modelling

The NAVITA approach attempts to remedy these problems as follows. NAVITA uses two separate diagrams for this modelling, Middle-level Functionality Diagram (MFD) and Lower-level Functionality Diagram (LFD). MFD is a simple diagram in which the granularity is constant, while LFD is used to describe detailed breakdown of the processes from MFD. MFD is a user-oriented view of the system functionality, while the LFD is more analyst-oriented.

9.3 Middle-level Functionality Diagram: Modelling Concepts

The main concepts used in MFD are:

- System Boundary
- Actor
- Functionality Unit
- Interaction

Generally UML notations which have been freely modified, as in SELECT Perspective and other methods, are used in this modelling. The UML version referred to here is version 1.3.

9.3.1 Actor

See Section 8.3.2.

9.3.2 System Boundary

See Section 8.3.1.

9.3.3 Functionality Unit

In NAVITA, the concept of functionality unit replaces use case for the reasons explained in Section 9.2.1. Like use cases, functionality units are representations of functional requirements describing what the user wants the system to do. From the business point of view, a functionality unit refers to a complete action carried out by the system to support an Elementary Business Process (EBP). SELECT Perspective defines an EBP as 'an atomic unit of work done by a person at a place at a time' (Allen and Frost, 1998). A functionality unit only includes the computerised aspects of an EBP and excludes the manual processes. Therefore, depending on the nature of the system, either a partial or a complete EBP may make up a functionality unit. This notion is similar to a 'function' in structured methods, and a complete use case in OOSE (Jacobson et al, 1992). A functionality unit, like a use case, is represented by an oval; see Figure 9.4. The key difference between a use case and a functionality unit is the granularity. With use cases, the granularity is variable; with functionality units, it is fixed. Functionality units have some important qualities that are explained below.

9.3.3.1 Granularity

In terms of granularity, a functionality unit could be compared with a function or an event response in Yourdon's terms (Yourdon, 1989). Each functionality unit makes sense in business terms and is easily traceable to business processes, which often means using meaningful terms such as Record New Customer, not Create New Customer Object. A functionality unit must therefore reflect a meaningful business operation. There should be a general one-to-one correspondence between a business operation and a functionality unit. Functionality units cannot be low-grained. For instance, Enter PIN in the ATM example cannot be a functionality unit because entering PIN itself does not constitute a complete business task since customers only enter PIN as part of a larger, complete task such as Withdraw Cash. Functionality units could neither be higher-grained; for example Counter Operations, in the bank example, is not a functionality unit because it contains many complete tasks such as Open Account, Transfer Money.

9.3.3.2 Atomicity

Each functionality unit is atomic in the sense that when applicable it is used in its entirety or not at all, it cannot be left half-done. One either does or does not withdraw cash. There cannot be a situation where the account is debited and yet cash is not dispensed to the customer. A functionality unit may be used independently of other functionality units without leaving the system in an inconsistent state.

9.3.3.3 Execution Time

A functionality unit does not take long to be executed, perhaps a few seconds or minutes, not hours or days. For example, it should take a few minutes to complete the functionality units Withdraw Cash or Borrow a Book.

9.3.3.4 Optionality and Immediacy

Functionality units are generally used in a group, which often stretches and obscures their limits. For example, Record New Order may sometimes require Record New Customer. Increased flexibility of user interface technology allows users to add new customers on the go, just when the order is about to be recorded. In such cases, questions can be asked whether to model them as a single functionality unit or multiple functionality units and why. NAVITA recognises that there are two aspects

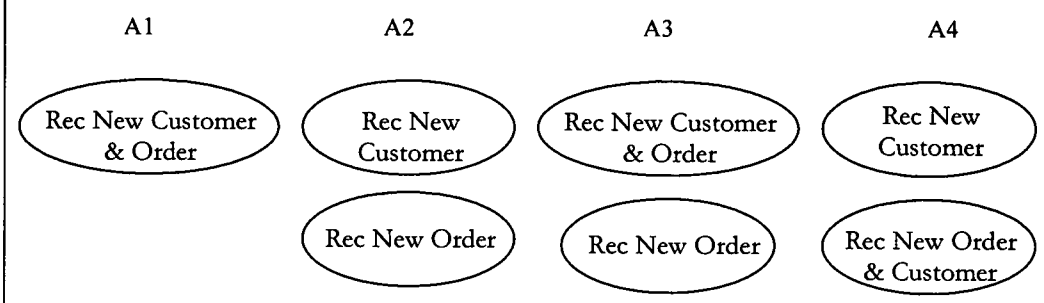
of functionality units that need to be looked at in such cases: optionality and immediacy. Optionality deals with the functional dependencies between the functionality units and immediacy, with the length of possible time gap between them. In this example, the following specific questions can be asked.

Q1. Does adding a new customer always and immediately lead to recording a new order?

Q2. Does recording a new order always and immediately require adding a new customer first?

There are four possible logical outcomes to these questions.

Figure 9.1 Four Possible Outcomes



* To avoid confusion, Record New Customer & Order can be renamed as Record First Customer Order, etc.

A1. Answers to both Q1 and Q2 are positive. It means that they are functionally and temporally inseparable, and therefore they ought to be regarded as a single functionality unit.

A2. Answers to both Q1 and Q2 are negative. It means that they are functionally and temporally separate, and hence they ought to be modelled as two distinct functionality units.

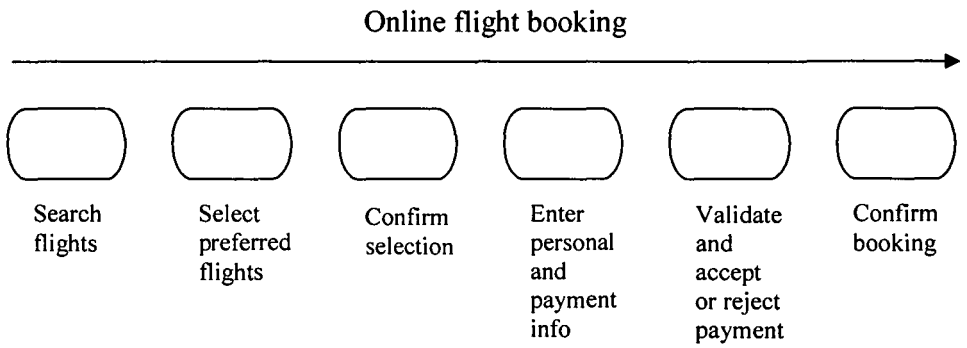
A3. Answer to Q1 is positive and Q2, negative. In this case Record New Order is functionally dependent on Record New Customer and the reverse is not true. The first thing that can be said then is that Record New Order is a functionality unit in its own right because Q2 is negative. Record New Customer must also be modelled in such a way that Record New Order is part of it.

A4. Answers to Q1 negative and Q2, positive. Record New Customer is a separate functionality unit and is part of Record New Order.

Further Examples

Take as an example the case of online flight booking processing. There are a number of stages the applicant has to go through before an application is processed.

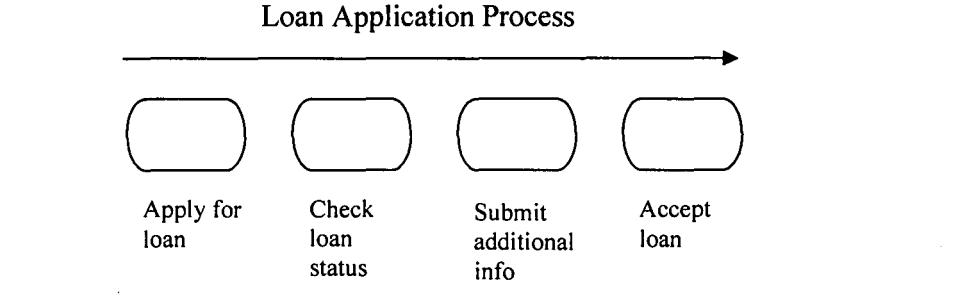
Figure 9.2 Low-grained processes of a functionality unit



Throughout this whole process, the user may stop and quit the process, but until the booking is confirmed at the last step, the state of the system or its database does not change permanently during the processing. There can be situations where the system remembers a long sequence of interactions with the user during stages before a commit is made, as in this example. The system seems to have changed its state during the data entry process but these are not permanent. That is, if the process is aborted, these temporary changes will be undone and the system will be put back to the state it was in before. Hence it is essentially a single functionality unit.

In a loan application process situation, the outcome may be different. The following example is adapted from Armour and Miller (2001).

Figure 9.3 Functionality units making up a larger process

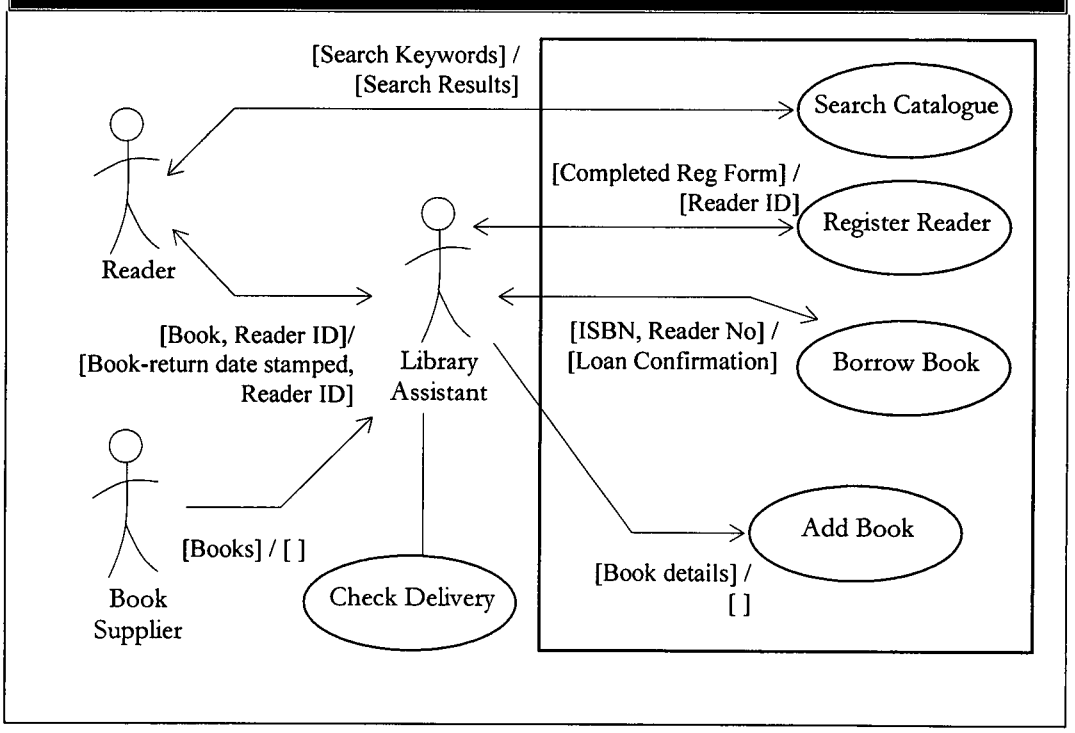


Do they make up a single functionality unit or many functionality units? Why? Armour and Miller (2001) suggest that because they achieve certain goals, they are separate use cases. From NAVITA's point of view, it is so because each of these steps is atomic and cannot be carried out partially. Each step has the right level of granularity in terms of its execution time and cannot last very long although the whole process would. Each step may take place days or months apart. Furthermore, after each step, the system is in a valid state.

9.3.4 Interaction

See Section 8.3.4.

Figure 9.4 A MFD for LibrInfoSys showing some functionality units



9.4 Middle-level Functionality Diagram: Modelling Process and Technique

Functionality modelling requires user participation in the process. General guidelines for the development of the diagram are as follows.

MFD Step – 1 Produce a MFD based on the interactions and some processes identified in the context diagram produced in CD Step 1 and CD Step 2, described in Section 8.4. If a business process is

supported by the system, it strongly indicates that it is a functionality unit. A complete set of interactions between an actor and the system, or the system and another system is also suggestive of a functionality unit. User's participation and understanding may be helped by the use of techniques such as JAD (Wood and Silver, 1995), paper prototyping, or even discussion around similar applications. For each functionality unit, add actor(s) who use(s) it, or the system(s) it/they interact(s) with. Also indicate the input and output information for each interaction.

The interaction [Search Keywords] / [Search Results] in the context diagram in Figure 8.6, for example, signifies that there is a process that deals with searching the catalogue, represented in Figure 9.4 by the functionality unit Search Catalogue. Other appropriate functionality units are also added to the diagram.

MFD Step – 2 Ensure that there is consistency between the context diagram the MFD in terms of the actors, external systems, and their interactions with the system. This may often call for the adjustment and expansion of the information provided in the context diagram. Eliminate the functionality units, manual processes and actors that are not relevant as a result of the changes.

The actors and their interactions with the system in the context diagram in Figure 8.6 and MFD in Figure 9.4 are identical in this case. If this is not the case at first, attempts must be made to make them consistent.

MFD Step – 3 Document the MFD using the chosen documentation standard such as the one suggested by NAVITA. Whatever standard is used, it is important that at least actors and functionality units are described in textual format. NAVITA documentation templates for Actor and Functionality Unit are provided in the following

section.

9.5 Middle-level Functionality Diagram: Documentation

The Middle-level Functionality Diagram is documented mainly by describing the actors, functionality units and interactions. NAVITA suggests the following documentation templates.

9.5.1.1 Documenting Actors

See Section 8.5.1.

9.5.1.2 Documenting Functionality Units

Documentation of functionality units includes pseudo-code style descriptions and the main scenarios in the functionality unit.

Figure 9.5 A Simple Template for Documenting Functionality Units

Functionality Unit No: ---------

Description: ----------

Step 1 : ----------

Step 2 : ----------

Relationships to manual processes (if any): ----------

Constraints: ----------

9.5.1.3 Documenting Interaction

See Section 8.5.1.3.

9.5.1.4 Additional Information

With larger systems, extra information about the diagram, such as when it was first created, who created it and when it is last updated, will be useful. A text box can be attached to the diagram, describing the date of creation, the author of the diagram and date of last update. This information can easily be extended as necessary.

9.6 Lower-level Functionality Diagram

This diagram shows the breakdown of each complex functionality unit, cross-references between functionality units, their dependencies and common elements. A number of candidates are considered for this diagram, in particular, Jackson Structured Chart (Goodland and Slater, 1995) and UML activity diagram. The latter is chosen as a candidate over the former due to relative simplicity of the diagram when describing complex flows; see a demo in the Appendix V. The NAVITA diagram itself is essentially a UML activity diagram with some modifications.

9.7 Lower-level Functionality Diagram: Main Concepts

The main concepts used in LFD are:

- System Boundaries, Actor and Functionality Unit
- Start and End
- Activity
- Flow (Sequence, Selection and Iteration)
- Swimlane
- Synchronisation
- Input and Output Data

9.7.1 System Boundaries, Actor, and Functionality Unit

See Sections 8.3.1, 8.3.2 and 9.3.3 for discussions on System Boundaries, Actor and Functionality Unit respectively.

9.7.2 Start and End

Like in UML, the beginning and end of the sequence of activities are indicated by a black circle and a bull's eye, as shown in Figure 9.6.

9.7.3 Activity

Activity represents any action, process, step or task of any granularity lower than or equal to that of functionality unit. Therefore, activity diagrams can be hierarchical. Diagrammatically activities are symbolised by a rounded rectangle, as shown in Figure 9.6.

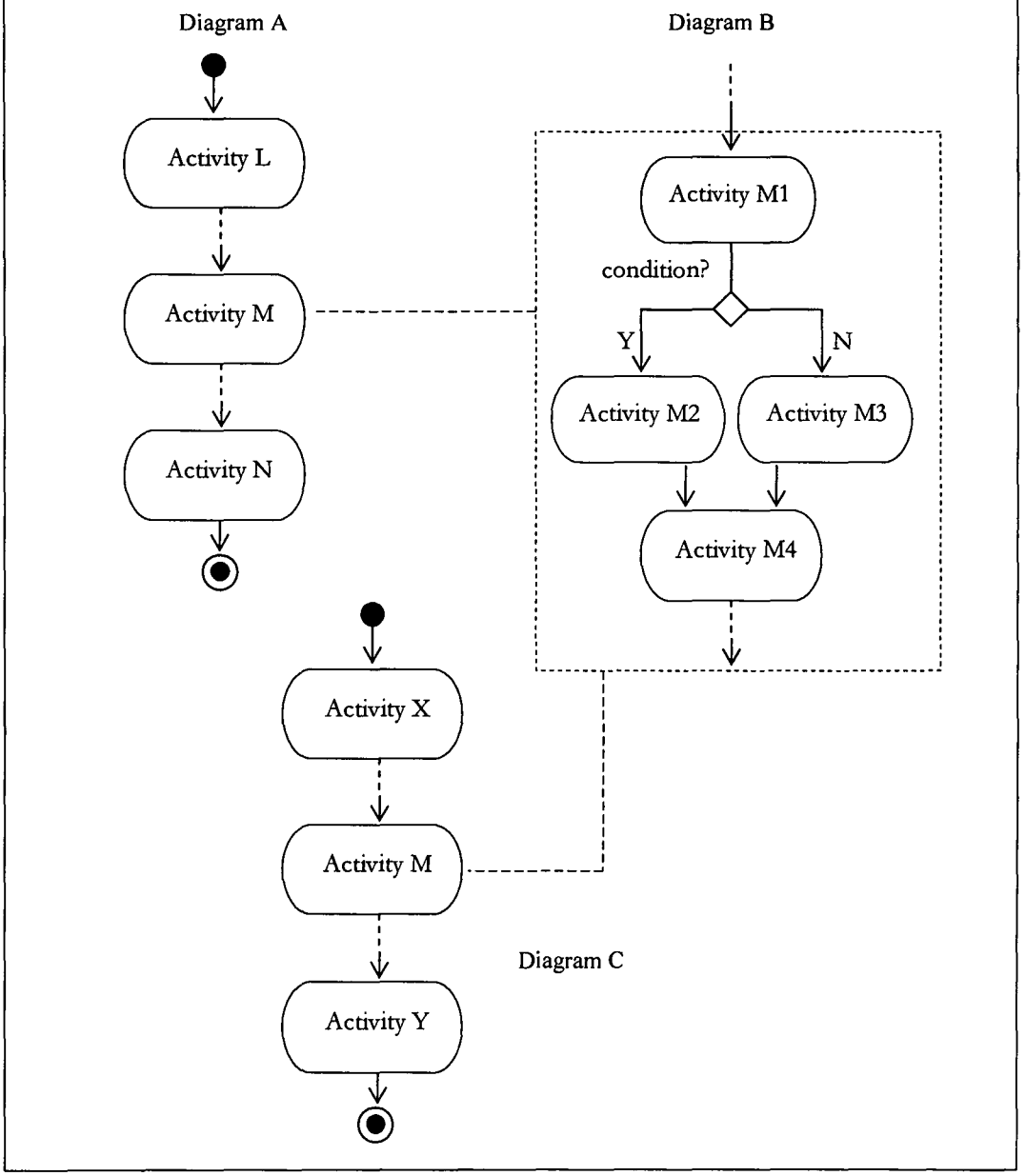
9.7.3.1 Hierarchical Activities

Activity diagrams can also be used to show further breakdown of complex activities in a hierarchical manner as shown in Figure 9.6.

9.7.3.2 Common Activities among Functionality Units

Nested hierarchical diagrams are also used to show common elements across functionality units, as in Figure 9.6.

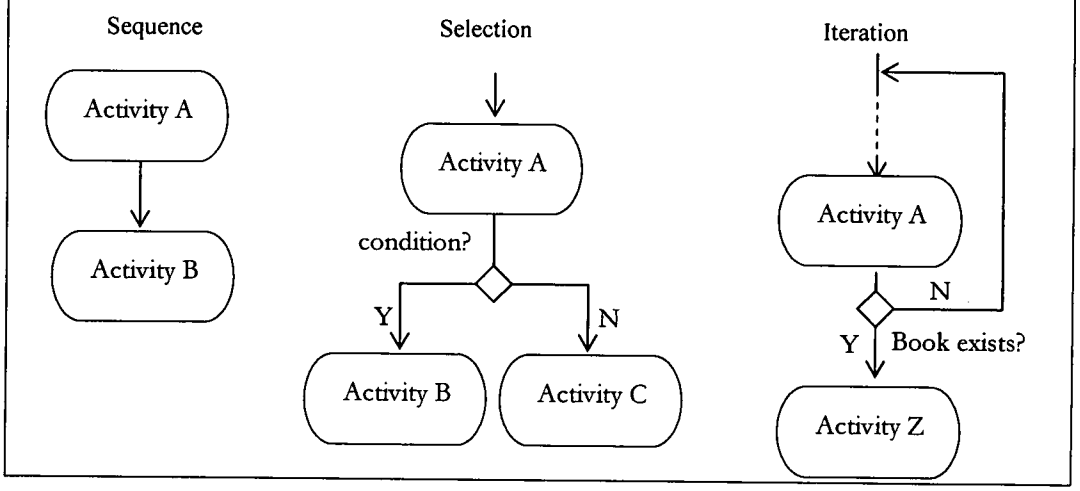
Figure 9.6 Hierarchy and Common Activities in LFD



9.7.4 Flow (Sequence, Selection and Iteration)

Sequential flow of activities is symbolised by a clear headed arrow indicating the direction of flow. A diamond represents the conditional branching, usually a binary selection. Multiple conditional branching is also possible, in which case the valid condition is written along the appropriate arrow; in such cases, the conditions must be mutually exclusive. The diamond symbol is also used to indicate conditional repetition of certain interactions. The condition may come either at the beginning or end of the series of interactions to be repeated. It can therefore cater for both While-Do-Until and Repeat-Do-Until types of loop.

Figure 9.7 Sequence, Selection and Iteration



9.7.5 Swimlane

UML Specification v1.5 (OMG, 2003) provides the following definition of swimlane.

> Actions and subactivities may be organized into swimlanes. Swimlanes are used to organize responsibility for actions and subactivities. They often correspond to organizational units in a business model.

The class may mean either an actor or system in this case. Swimlanes are represented by dotted horizontal lines in the diagram.

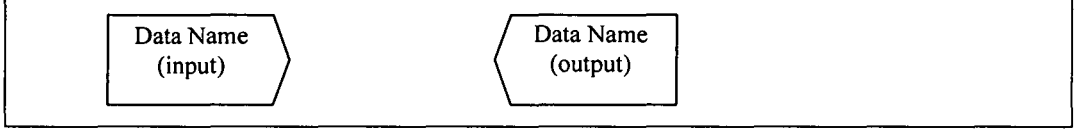
9.7.6 Synchronisation

UML Synchronisation bars are used to show the points of diverging and converging flows of parallel activities.

9.7.7 Input and Output Data

Input and output data do not exist as a concept in UML, but are represented in NAVITA by either rounded rectangles (like UML Activity) or preferably, pentagons. If rounded rectangles are used, it is important to denote each of them with a stereotype, either <<input>> or <<output>> so that they will not be mistaken for activities. If pentagons are used, it is not necessary to use the stereotypes, but advisable for the simple reason of readability. A pentagon pointing right indicates some data coming from the interacting actor to the system, while another pentagon pointing left indicates some data going out from the system to the interacting actor. Each data item is given an appropriate name.

Figure 9.8 Input and Output Data



9.8 Lower-level Functionality Diagram: Modelling Process and Technique

LFD Step – 1 Produce an LFD for each complex functionality unit. Complexity of a functionality unit is ascertained from the textual description of the functionality unit in the documentation. Nest the LFD diagrams if activities are too complex, long or a set of activities are shared between functionality units. Development of LFD will help clarify the system's boundary, hence its functionality, by enabling the analysts to look closely at the processes that sit along the boundary. User participation will help decide which parts of the processes are inside the system, and which, outside of it, i.e. manual processes.

LFM for Register Reader in Figure 9.9 shows the detailed steps and interactions, including physical interactions, between actors

and the system. Activities in Reader and Library Assistant swimlanes indicate manual processes and the activities in System swimlane are computerised activities. Similarly, interactions between actors can be physical while those with the system must be logical.

LFD Step – 2

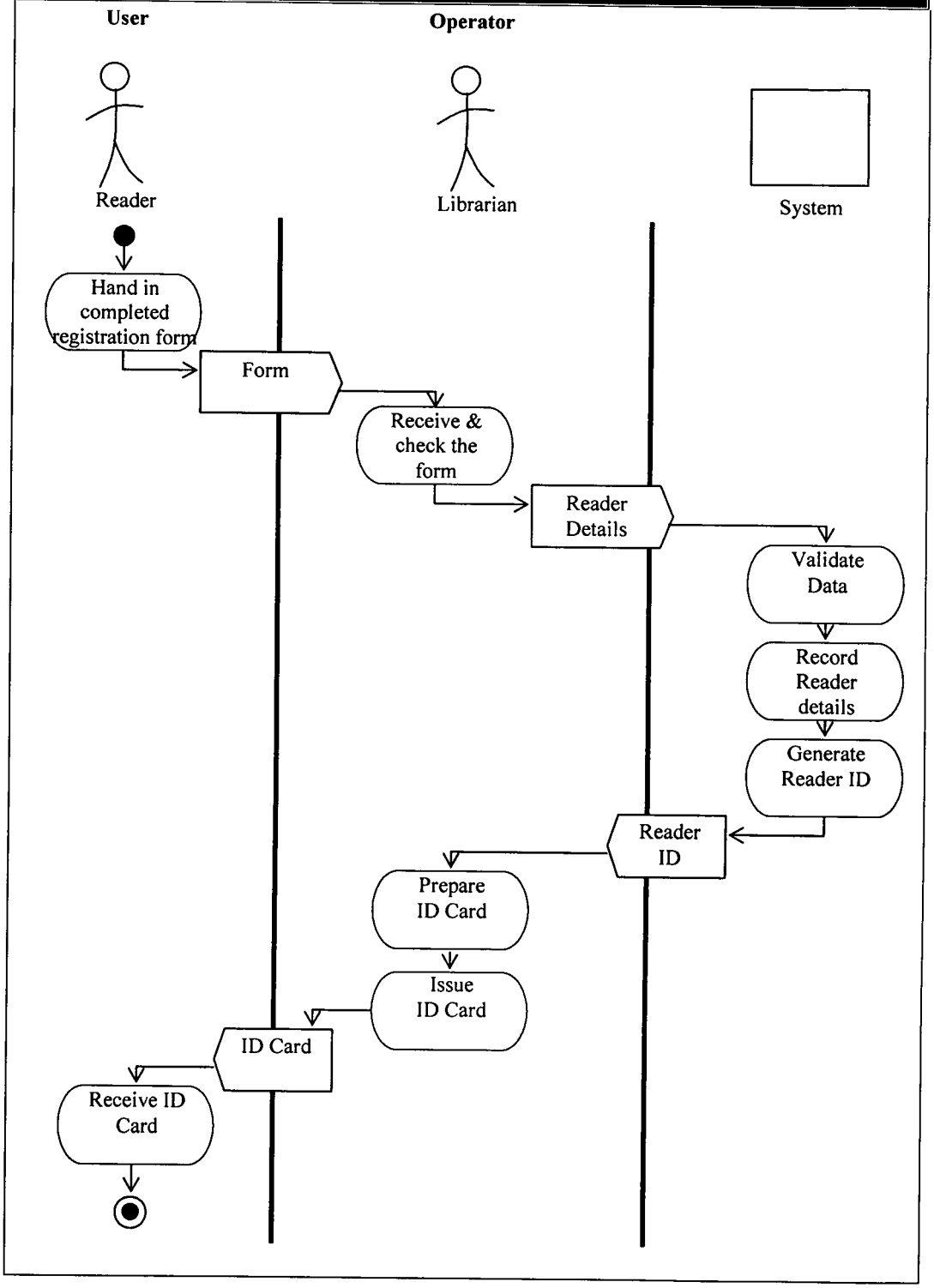
Maintain the consistency of cross-references between the business processes, context diagram, MFD, MFD documentation and LFD. Functionality units in MFD must reflect business processes. Actors and interactions in context diagram and MFD must be consistent and documentation of functionality units must be in line with the activities in LFD. Inputs and outputs in LFD, MFD and context diagram must also be in agreement.

The interaction between the operator actor and the system includes Reader Details as input and Reader ID as output, which is consistent with interactions in MFD and the context diagram. The manual processes of Reader and Library Assistant are not shown in the early diagram, which is permissible if they are deemed insignificant to the system's main operation. In this example, the Prepare ID activity is carried out by the Library Assistant on the assumption that the card is manually prepared. If the system is to produce it automatically, it will be inside the system's swimlane. That's how the diagram is used to explore the system's boundary.

LFD Step – 3

Revise both the MFD and the context diagram in the light of detailed knowledge gained from LFD and also their associated documentations. In this case, the manual processes of Reader and Library Assistant may be added to the diagram.

Figure 9.9 Lower-Level Functionality Model for Register Reader



9.9 Lower-level Functionality Diagram: Documentation

NAVITA does not suggest any specific template for documenting LFD because it is, in most situations, a diagram that is detailed enough to explain itself.

9.10 Development Process

This modelling is used in Stage 1 – Feasibility Study and Stage 2 – Business Study and Requirements Investigation. As explained, all these diagrams and documentations may not be necessary for all scenarios of development. In cases where there is a good chance of reusing an entire application, the MFD and its documentation is likely to be useful. If the nature of application and availability of existing applications is such that the MFD and its documentation do not provide enough detailed information, LFD may be applied limitedly in Step 1 and Step 2. In Stage 4 – Detailed Requirements Analysis, however, this modelling is applied in full scale.

9.11 Software Architecture

Since the diagrams in this modelling help define the user's requirements, their impact on the software architecture is crucial in the sense that functionality qualities of individual components and the application as a whole must be in agreement with the diagrams produced here.

Chapter Ten

NAVITA System Modelling
System Interaction Modelling

10.1 Introduction

This chapter discusses modelling tools and techniques provided by NAVITA used in the analysis of the interaction between the system and its users.

Human-Computer Interaction (HCI) is a well established area of study in computer science. Those who take a close interest in the subject regard it as a design activity that is concerned with all aspects of usability 'from the abstract design of system scope, contents, and functionality, to the detailed design of the presentation and interactivity of the actual concrete user interface to the interactive system'. However, those from a more 'traditional' software engineering background regard HCI as less essential in system development (van Harmelen, 2001). User Interface Design (UID) in NAVITA will decidedly not deal with the many aspects of usability of HCI for two main reasons. First, arguably such advanced HCI issues as cognitive, ergonomics and accessibility merit separate research in their own right. Second, these issues are outside the scope of this research into CBSD methods with particular emphasis on system analysis and design. Having said this, it is well recognised that if done properly, logical User Interface Design can lend itself as an invaluable tool for the identification, analysis, communication and verification of users' requirements and system design. In this sense UID is absolutely crucial for system analysis and design, as well as for usability. ISO defines usability as follows (ISO, 1998):

> Usability is the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in use.

Therefore, UID modelling in NAVITA would exclusively focus on the basic visualisation of interface, logical data and interaction between the user and the system, leaving out the usability issues to HCI design methods (Shneiderman and Plaisant, 2004).

10.2 System Interaction Modelling

There are two diagrams used in this model, Logical Screen Layout (LSL) and User-System Dialogue Model (USDM).

10.3 Logical Screen Layout

Logical Screen Layout (LSL), produced for each functionality unit, is a loosely-structured visualisation of the static and simplified interface of the system which may have little correlation to the layout of the concrete interface, screen flow, menu and navigation. The main emphasis of the diagram is to capture the data communicated through the interface, i.e. inputs and outputs, and not the physical screens' design using a range of GUI objects offered by the programming languages, navigability (to other functionality units), user friendliness or similar usability issues. Usually, there will be a screen layout for each functionality unit. In complex cases, particularly those with web-based interface, there will be a need to have a few sub-screens within the screen layout of a functionality unit.

This diagramming could be supported by some simple prototyping. One can use paper-based prototypes which are quick, easy and inexpensive. This also serves the purpose of clarifying the users' requirements and their understanding of the system. In some cases, it is also desirable to have a 'real' prototype, i.e. computer based, with some kind of simulation, which is more effective in clarifying what the users want. Nevertheless, it must be remembered that the main emphasis is on the information that goes in and out, not the user interface design itself. Because it is a logical design, analysts need not be interested in the type of input/output mechanisms. In some cases, it would be necessary to indicate the presentation method, such as textbox, drop-down list or combo-box.

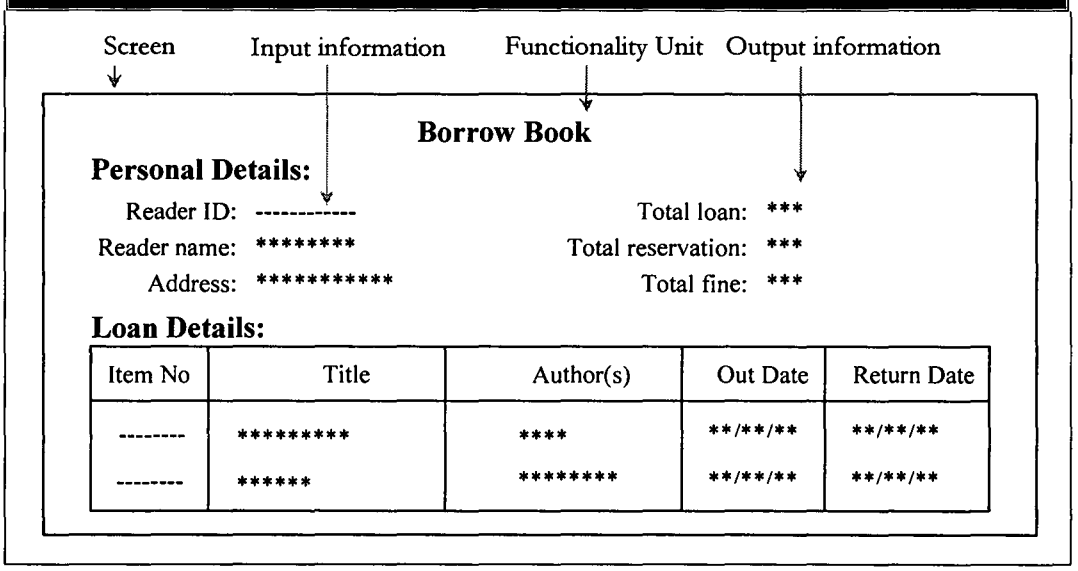
10.4 Logical Screen Layout: Modelling Concepts

The main concept used in this diagram is the Input and Output Fields.

10.4.1 Input and Output Fields

In NAVITA, Input fields are denoted by a dotted line next to them, and output fields, asterisks, as shown in Figure 10.1. Although the diagram is usually read from left to right and top down, the detailed sequences of interaction need not be emphasised in this diagram as they are explored in great detail in User-System Dialogue Model (USDM).

Figure 10.1 Logical User Interface Diagram



10.5 Logical Screen Layout: Modelling Process and Technique

This is a rather informal analysis, which requires extensive user participation. The main aim is to capture the static logical interface between the system and the user. Although this modelling can be done using pen and paper, it could also be supported by an automated form of prototyping. The main aim of these diagrams is to visualise the user-system interface; in particular the sort of information that the user will provide to the system and information the system provides to the user. Although it uses fairly informal notations, these diagrams are important. As this is a logical view, there is no need to emphasise the implementation-specific details of the interface or address non-functional issues such as navigability, usability and security, unless these concerns are critical to the functionality units.

LSL Step – 1 Start by choosing an appropriate title for the screen, usually the name of the functionality unit. Sketch out various data items that are likely to appear on the computer screen, paper report or other media of communication between the user and the system, with the help of users, the existing system, similar applications and documents. Input fields are noted by dotted lines and output fields by asterisks.

LSL Step – 2 Ensure that the input and output fields tie in with the input/output data in data flows of context diagram, MFD and LFD.

10.6 Logical Screen Layout: Documentation

NAVITA does not suggest documentation templates for this modelling since these diagrams are often self-explanatory and need no further descriptions. If necessary, documentation templates such as Data Dictionary (Goodland and Slater, 1995) can be used.

10.7 User System Dialogue Model

Whilst LSL shows the visualisation of the static and simplified interface, System Dialogue Model (USDM) models the dynamic and sequence-oriented interactions between the system and the user. Many OO methods, in particular UML, do not provide diagrams that exclusively focus on the input/output-based interactions between actors and the system. Interactions between the user and the system through messages shown in sequence and collaboration diagrams are often obscure and not detailed enough.

SSADM has I/O Structure Diagram that uses JSD Structured Diagram notations, which can present a great challenge to those who are new to it. The diagram tends to get cluttered if nested selections and interactions are introduced. As an example, one can compare the two figures in Appendix V.

It is clear from these figures that UML Activity Diagram has the similar concepts used in JSD Structured Diagrams. Furthermore, the UML figure shows the flow

more clearly and UML has more intuitive rules. However, UML activity diagrams lack the notions of input and output. In NAVITA, UML Activity Diagram notations are modified and adapted for this purpose. It is a free modification of UML because UML simply does not have anything equivalent to inputs and outputs of SSADM's I/O Structure Diagram.

10.8 User System Dialogue Model: Modelling Concepts

The main concepts used in this diagram are:

- Start and End
- Input and Output Data
- Sequence, Selection and Iteration
- Scenario

10.8.1 Start and End

See Section 9.7.2.

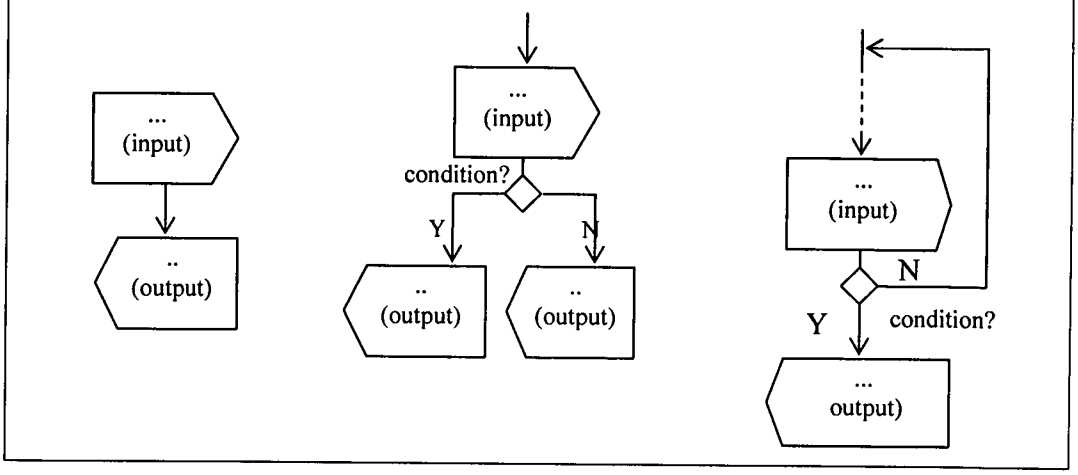
10.8.2 Input and Output Data

See Section 9.7.7.

10.8.3 Sequence, Selection and Iteration

A clear-headed arrow indicates the flow of the interaction; see Figure 10.2

Figure 10.2 Sequence, Selection and Iteration



10.8.4 Scenario

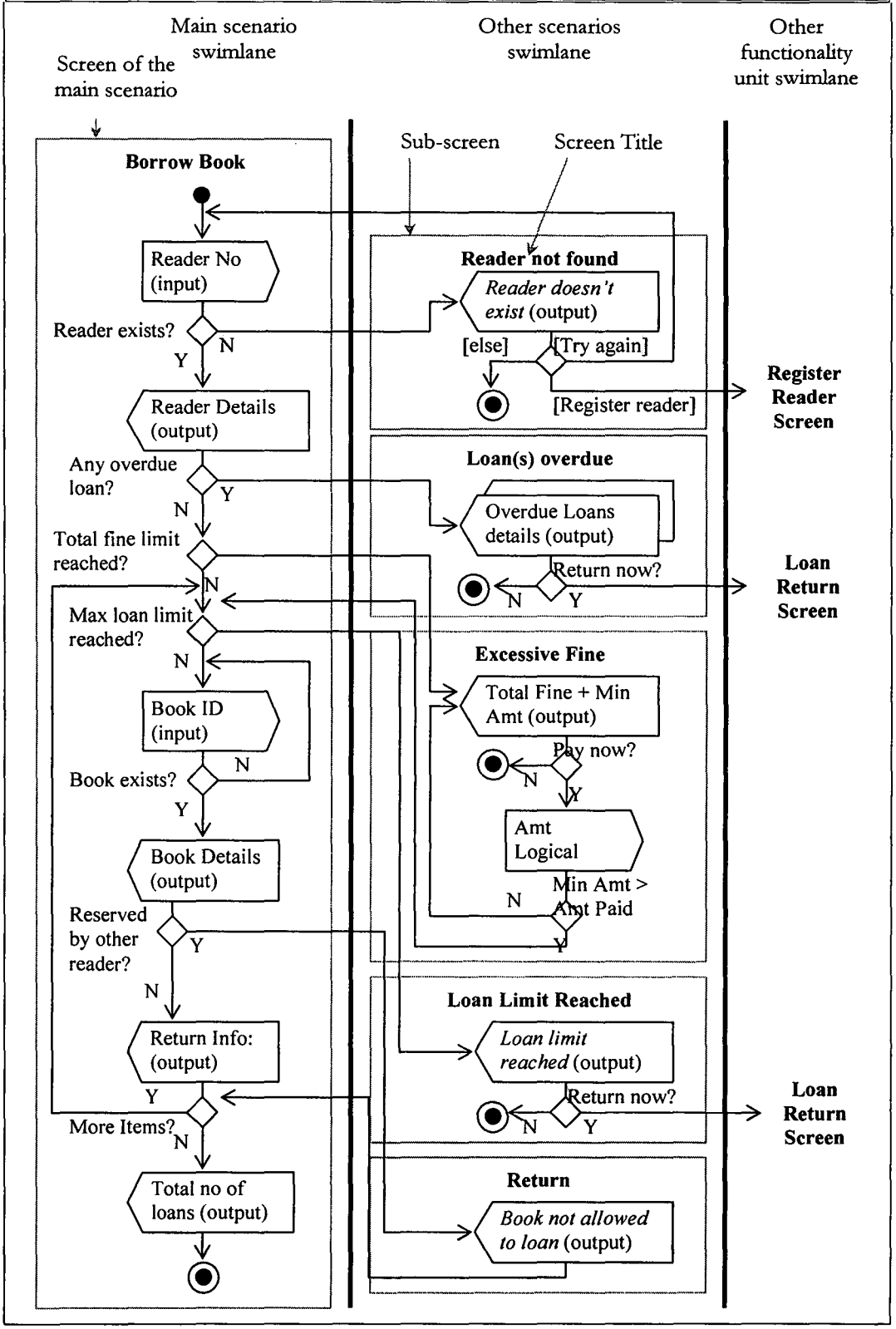
USDM Scenarios are represented by swimlanes described in Section 9.7.5, see Figure 10.3. The leftmost swimlane indicates the main successful course of interactions achieving the user's primary goal of using the functionality unit. There can also be other scenarios indicating other courses of interactions, shown in the next swimlane. USDM can be improvised to add further information about the diagram. For example, it can show how various screens will be organised. Figure 10.3 shows how the main screen and various sub-screens of a functionality unit, and screens of other functionality units, can be connected. Depending on the nature of navigations envisaged, flows to screens of other functionality units can be made to enable resumption at the point the flow quitted the original functionality unit. It must be emphasised that these extra details may only be necessary in specific circumstances, such as in cases of paper-based prototyping.

10.9 User System Dialogue Model: Modelling Process and Technique

USDM Step – 1 Work through IO fields in LSL to understand the dynamic sequence of interactions between the actor and the system, and depict those interactions in a USDM diagram. First show the primary sequence of interactions that make up the main scenario in the first swimlane. Then add other scenarios.

LSL for Borrow Book in Figure 10.1 shows that the process begins when the Reader ID is keyed in. Then the reader details need to be retrieved and displayed, which involves checking the reader record first. It is possible that the reader record does not exist for the ID. In this case a message would be shown and the system may ask the user to try again. However, the main scenario assumes that no such error would occur, and reader details would be retrieved. The entire sequence in the first swimlane therefore represents the successful scenario for the functionality unit, while boxes in the middle swimlane represent the possible exceptional scenarios. The rightmost swimlane represents the screens of other functionalities to which the user may wish to navigate.

Figure 10.3 USDM for Borrow Book



USDM Step – 2 Crosscheck the control structures of the LFD and USDM.

The control structure in of the Borrow Book LFD should match the control structure of the Borrow Book USDM in Figure 10.3.

10.10 Development Process

This modelling is mainly used in Stage 4 – Detailed Requirements Analysis and Stage 5 – Prototyping.

10.11 Software Architecture

LSL and USDM are vital parts of NAVITA for two main reasons. First, these models are used to encourage user participation in the development of system models, which helps ensure the external consistency of the specifications, and second, the USDM are important to identify the operations involved in component communication and the structure of the communication.

Chapter Eleven

NAVITA System Modelling
Information Modelling

11.1 Introduction

NAVITA Information Modelling is based largely on the traditional Entity Relationship Modelling and Class Modelling. This modelling uses a diagram and a cross reference matrix, namely Information Model and Functionality Entity Class Matrix (FEM).

11.2 Information Model: Modelling Concepts

It uses UML Class notations, where the main concepts are:

- Entity Class
- Attributes
- Relationships – Association, Aggregation and Composition
- Inheritance

11.2.1 Entity Class

As discussed in Section 6.4.2.6, entity classes are used not only to represent real-life objects, but their data structures too. At the early stage of development, NAVITA entity classes are operationless, like entity in Logical Data Structures, which can be defined as follows (Goodland and Slater, 1995):

> An entity is something of significance to the system about which information is to be held.

Only in later stages of development, after a decision is made to implement the system using an OO technology, these entities are translated into appropriate OO classes. UML Specification v1.5 (OMG, 2003) defines Class as follows:

> A class represents a concept within the system being modelled. Classes have data structure and behaviour and relationships to other elements.

11.2.2 Attributes

In the context of Information Modelling, attributes are regarded as properties of entity classes.

11.2.3 Association, Aggregation and Composition

UML Specification v1.5 defines association as follows:

> A binary association is an association among exactly two classifiers (including the possibility of an association from a classifier to itself).
>
> An association may represent an aggregation (i.e., a whole/part relationship). In this case, the association-end attached to the whole element is designated, and the other association-end of the association represents the parts of the aggregation. Only binary associations may be aggregations.
>
> Composite aggregation is a strong form of aggregation, which requires that a part instance be included in at most one composite at a time and that the composite object has sole responsibility for the disposition of its parts. The multiplicity of the aggregate end may not exceed one (it is unshared).

The specification suggests three kinds of associations: ordinary association, composite aggregate and shared aggregate. Generally speaking, aggregate associations denote whole-part relationships; shared aggregate allows 'part' objects to be shared by 'whole' objects, and composite aggregate does not. Composition is regarded as an aggregation with strong ownership. See Sections 2.5.4 Semantics and 3.47 Composition of the UML Specification v1.5 (OMG, 2003) for a detailed treatment of Association, Aggregations and Composition.

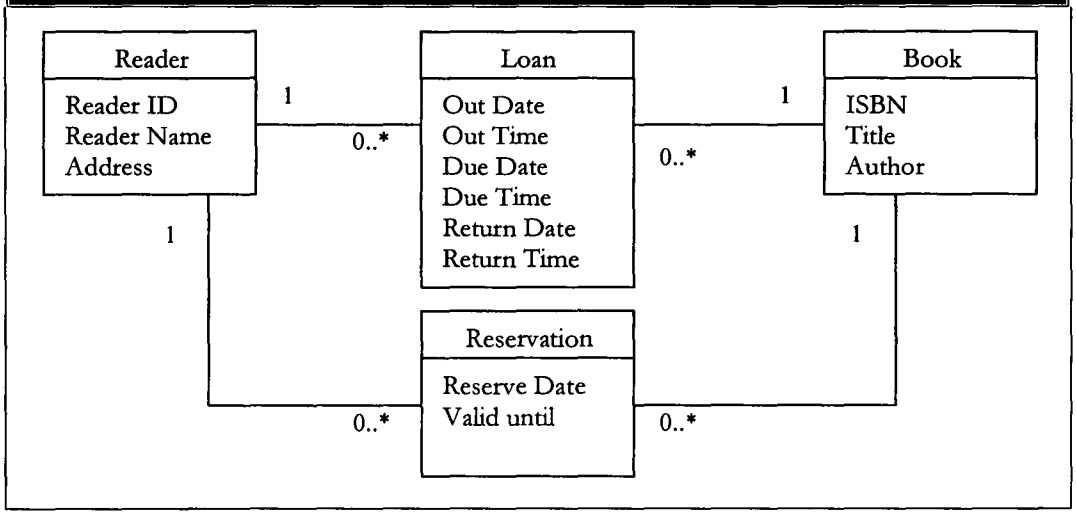
11.2.4 Inheritance

UML Specification v1.5 (OMG, 2003) often refers to this as generalisation/specialisation relationship between model elements.

Generalization is the taxonomic relationship between a more general element (the parent) and a more specific element (the child) that is fully consistent with the first element and that adds additional information. It is used for classes, packages, use cases, and other elements.

Inheritance is used only when the entities are translated into OO classes as a result of a design decision to implement the system using an OO technology; see Section 13.8.

Figure 11.1 A simple IM showing Entity Classes, Attributes and Relationships



11.3 Information Model: Modelling Process Technique

As in SSADM v4 (Goodland and Slater, 1995), NAVITA provides two starting points for the development of Information Model. The first attempt at the diagram can be called the top-down approach, in which entity classes are identified first, followed by their relationships and attributes.

IM Step – 1 Identify candidates for entity classes from the descriptions of functionality units by analysing the keywords in them. Each class should be of importance to the system, have multiple attributes, include multiple instances and each instance may be unique. Attributes describe entity classes; for example, reader ID, name and address describe Reader.

IM Step – 2 Specify the relationships (association, aggregation etc) between these

classes and the cardinalities of relationships. Each relationship must reflect the real world relationship such as reader borrows a book or the data structure, such as a title has many book copies. Cardinalities should mirror the business constraints, such as a reader cannot borrow more than a certain number of books at a time.

IM Step – 3 Inheritance relationships are identified in two ways. Instances of some entity classes belong to different categories, such as different types of readers. Some entities may have common attributes, for example a student and a reader may have a number of attributes in common. In both cases, inheritance relationships are required.

This would give the analyst the first overview of the entity classes in the system. This diagram needs to be neither complete nor accurate at this stage.

The second strand of development of the same model starts from a different and a more concrete point. The input and output fields in LSL (Section 10.3), and to some extent, USDM (Section 10.7) and Functionality Modelling (Chapter 9) too, provide a detailed set of attributes that will be grouped into entity classes.

IM Step – 4 Gather attributes from LSL of each functionality unit. Ensure that LSL has already been crosschecked with USDM, context diagram and functionality modelling. Remove duplicated and extraneous classes and attributes.

IM Step – 5 Identify entities from these attributes through informal analysis or using Relational Data Analysis or Normalisation (Goodland and Slater, 1995).

IM Step – 6 Specify associations between the entity classes producing a fragment of a global information diagram.

IM Step – 7 Crosscheck the diagram fragment with the initial information diagram and ensure consistency between the two diagrams.

IM Step – 8 Repeat IM Step 3 to IM Step 6 for each functionality unit.

11.4 Documenting Information Model

11.4.1 Documenting Entity Classes and Attributes

Entity classes and attributes can be documented using the following template.

Figure 11.2 Template for documenting Entity Classes and Attributes

Entity Class Name: ---------

Description: ----------

List of attributes and data types: ----------

11.4.2 Documenting Relationships

It is not always necessary to document the relationships. However, if the situation calls for it, the following convention can be adopted.

Figure 11.3 Template for Documenting Entity Class Relationships

Relationship Name: ---------

Description: ----------

Optionality and Cardinality: ----------

Other constraints: ----------

11.4.3 Extra Information

A plain text box describing when the diagram was first created, who created it and when it was last updated, can be attached to the diagram. This information can easily be extended as necessary.

11.5 Functionality Entity Class Matrix (FEM)

This matrix is largely borrowed from SSADM and is adapted for this method. The need for such a matrix is highlighted in our paper (Bielkowicz and Tun, 2001). This matrix shows the correlations between functionality units, listed in rows, and entity classes, listed in columns, in terms of the various effects functionality units have on entity classes. The effects are denoted as follows:

- I for inserting or creating a new instance of the entity class
- M for modification of some attribute value(s) of an instance of the entity class
- D for deletion of an instance of the entity class
- R for reading attribute value(s) of an entity class
- L for linking two instances of one or two entity class(s). Ls are always used in pairs; each pair can be numbered for clarity if there are many of them
- C for cutting links of two instances of entity classes, again used in pairs; each pair can be numbered for clarity if there are many of them
- * is used to denote that the effect applies to more than one instance of the entity class; for example R* indicates reading from multiple instances of an entity class
- [ ] is used to denote optional effect(s)

Figure 11.4 A Simple FEM for LibrInfoSys

Functionality Units	Entity Classes					
	Reader	Loan	Reservation	Title	Copy	Author
Register Reader	I					
Add Book				IL	IL	
Add Copy				L	IL	
Borrow Book	LM	ILL	[M]		L	
Show reader's current loans	R	R*				
...						

11.6 FEM Modelling Process and Technique

FEM Step – 1 List all entities from the global Information Model, and all middle-level functionality units in columns and rows of FEM respectively.

FEM Step – 2 Identify effects of functionality units by looking at whether each functionality unit creates or deletes instances of entity classes and

their relationships, or modifies attribute values of them.

FEM Step – 3 Ensure that each entity class has at least one functionality unit to create, one to delete its instances; often there are other functionality units to modify its attributes and links. In some cases, such as archiving, it may be acceptable to have an entity that is never deleted. Otherwise, consider adding other functionality units or removing the entity class. For each functionality unit there is at least one entity affected. Otherwise consider removing.

11.7 Development Process

IM Step 1 and IM Step 2 of this modelling is used in Stage 1 – Feasibility Study and Stage 2 – Business Study and Requirements Investigation and IM Step 3 to IM Step 7 are used in Stage 4 – Detailed Requirements Analysis.

11.8 Software Architecture

IM modelling is an essential part of logical component specification (see Section 12.6); this model is used to define the static structure of the system as well as part of the interfaces of components. IM, by validating the functionality model through the FEM, helps ensure that the architecture satisfies the functional requirements.

Chapter Twelve

NAVITA System Modelling
Architectural Analysis

12.1 Introduction

The main aim of NAVITA architectural analysis is to identify the services used by logical boundary components that logical business components provide. The main product of this analysis is a precise specification of these services with minimal references to the implementation technologies of physical components. The discussion so far has concentrated on the analysis of the system as perceived largely by its users, or the external view of the system. Now the focus turns to the internal details regarding the logical structure of the system, its components, and their interfaces and interactions.

NAVITA System Architectural Model, as discussed in Section 6.3, assumes that, at this logical level, for each functionality unit, there would be a boundary component and a business component. Interactions between these two components realise the functionality unit. This separation of boundary and business components allows analysts to explore the demarcation of responsibility between the two types of component, leading to a clear understanding of what a component needs to do if it is going to provide or use the service. At this stage, the best scenario that analysts hope for is finding components that satisfy some or all of these services without needing to design and implement the components. Therefore, what is necessary at this stage is some clear specification(s) of interface of services to be offered by components of the system.

12.2 Architectural Analysis: Diagrams and Concepts

With Logical User Interface Diagram and User-System Dialogue Model, analysts attempt to analyse in detail the interactions between the user and the system. Architecture analysis, on the other hand, is concerned with the interactions between boundary component and business component of each functionality unit. The interface between the two components is specified by the Protocol Model, which defines the possible way(s) in which two components interact. Fragments of Information Model are then developed for each Protocol Model to establish the underlying data structure within the business component. The main models used in this analysis are:

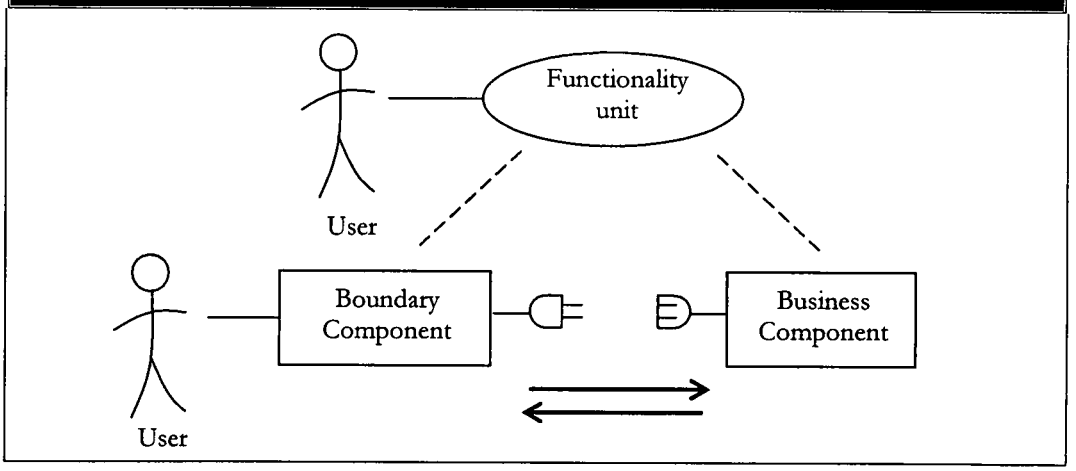
- Protocol Model
- Information Model fragment

These models contribute towards the production of logical component specifications, the main output of this architectural analysis.

12.2.1 Protocol Model

The protocol model is used to analyse the interfaces between two collaborating components by looking at the way(s) in which the two components communicate to realise the functionality unit; see Figure 12.1. Generally, the logical business component will provide a service used by the boundary component. This presents an opportunity to analyse each functionality unit in isolation.

Figure 12.1 Boundary Component and Logical Business Component realising a functionality unit



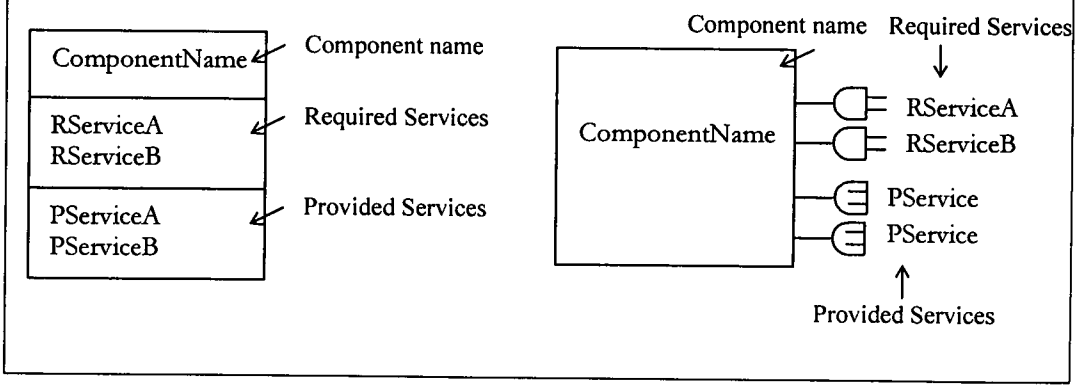
The main concepts used in this diagram are:

- Logical Component and Logical Component Interface
- Service
- Operation
- Sequence, Selection and Iteration.

12.2.2 Logical Component and Logical Component Interface

The notion of component used in NAVITA is explored in Chapter 6. There are two types of components at this logical level: business components and boundary components. Notationally, a component can be drawn using two different conventions. In one convention, it can be drawn like a UML class; the component name is written in the top of the three compartments, while the other two compartments list services the component provides and requires. In the other convention, required and provided services are represented by plug and socket icons, as shown in Figure 12.2. These three compartments make up the interface of the component.

Figure 12.2 Component and Services



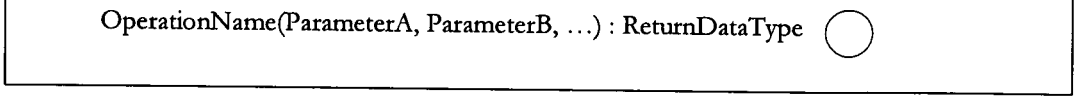
12.2.3 Service

A service can be defined as a set of operations provided by a component, which constitute a meaningful business operation. Detailed discussion on service is given in Sections 6.2.3 and 6.5.1.2. Services are either written in the compartments or written next to the plug-socket icons.

12.2.4 Operation

Component services are provided through operations. Service operations are similar to class operations in terms of their makeup. An operation has a name, may take some parameters and has a return data type (OMG, 2003). Operations in NAVITA Protocol Model are represented by circles with the name of the operation next to it.

Figure 12.3 An operation and control structures



12.2.5 Flow – Sequence, Selection and Iteration

See Section 9.7.4.

12.3 Protocol Model: Modelling Process and Technique

PM Step – 1 Assume that each functionality unit is realised by collaboration between a logical boundary component and a logical business component. Strictly speaking, the collaboration is between instances of the two components rather than their types; however, for brevity, instances are generally referred to as components, and if the distinction is necessary, it is made clear in the discussion.

In Figure 12.4, two components BorrowBookUI and BorrowBook are created for the functionality unit Borrow Book.

PM Step – 2 Identify the operations the business component has to provide to the boundary component from LFM, USDM and also system functionality documentation. By going through the steps in USDM, for example, analysts can question what the system will have to do at each step. Each atomic task the system has to carry out is represented by an operation. The parameters of the operations are determined from the interactions. The flow of these operations should reflect the structure in LFM, USDM and the descriptions. Produce a protocol model based on the information gathered.

Figure 12.4 Protocol Model for Borrow Book functionality unit

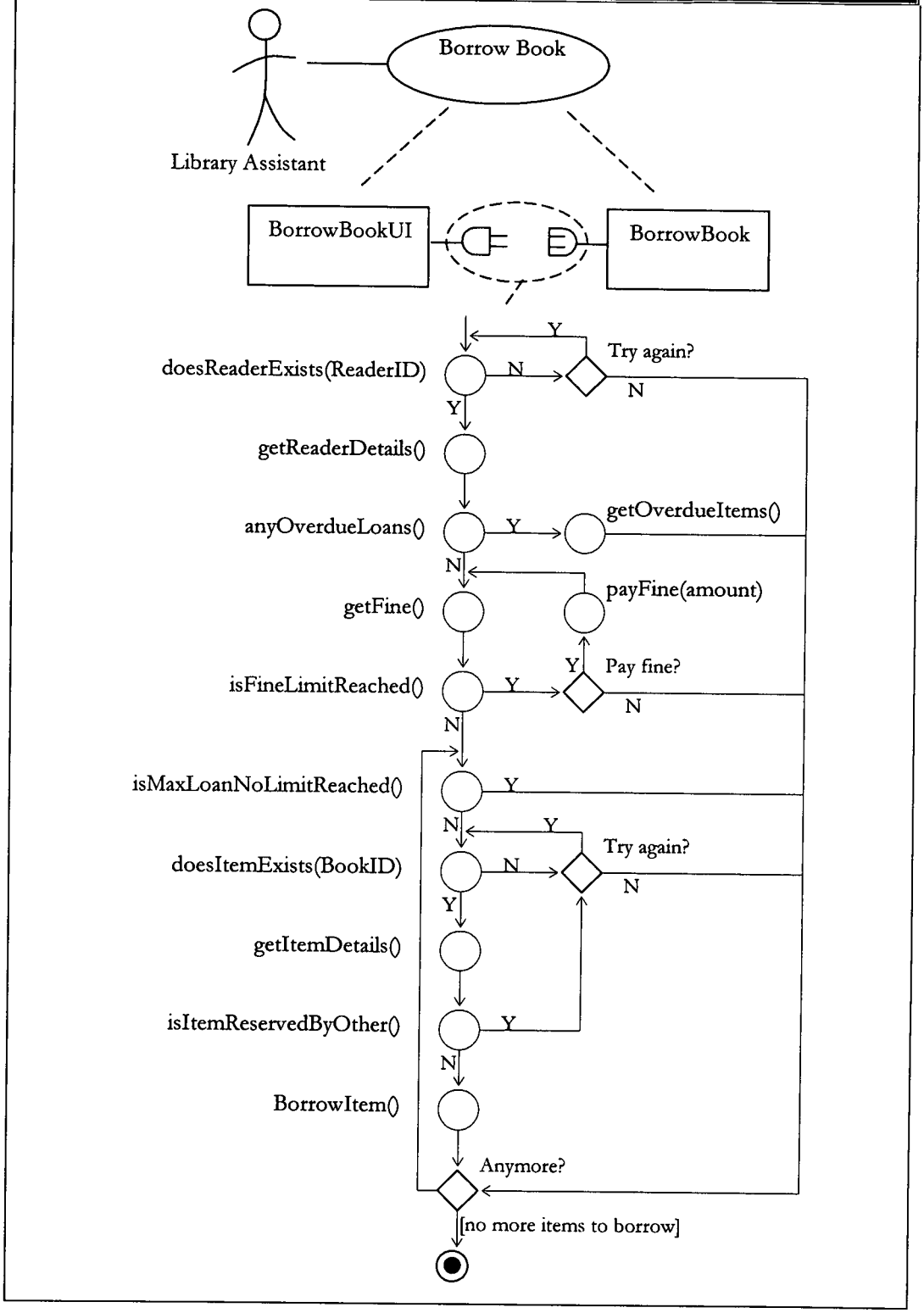


Figure 10.3 shows the USDM of Borrow Book. After the input reader number is accepted by the boundary component, the system

has to determine whether the reader information is stored inside the system, which the boundary component cannot determine by itself. The boundary component will have to consult the business component by sending a message or operation call to it. This is represented by the operation doesReaderExists(ReaderID) in the protocol model in Figure 12.4.

By sending this operation to the business component, the boundary component is in effect creating a new 'session' or instance of communication between them. It means that, from then on, the two components have committed themselves to the completion of the entire sequence of operations. Generally speaking, either of the components may quit the session at anytime, and if the state of the system has not been updated permanently, which tends to happen towards the end of the interactions, all temporary changes[8] will be undone and the session unsuccessful. Once the session has started, both components will remember the status of the session in terms of the stage they are at in the protocol model, and they will proceed according to it. In the example, after the operation doesReaderExists(ReaderID) is called, and details of the reader record are found to exist in the database, these details then need to be displayed, according to the USDM. Again, the boundary component needs to get the information from the business component by sending another operation getReaderDetails(). If the reader record is not found in the database, this protocol requires that the system checks with the user to see whether he or she wishes to try inputting an ID, or rather quit the process.

It should be noticed that the second operation in the protocol model, getReaderDetails(), does not carry the reader ID as a

[8] In some cases, it may require committing changes in stages. For example, if the session is stopped after some fine is paid, it should not be undone. Similarly, quitting in the process of borrowing a book should not cancel the earlier loan etc.

parameter. This is so because once the reader has been identified in business component by the earlier operation, the boundary component does not need to pass the same parameter when accessing information relating to it. If in doubt, the parameter can be passed again.

After displaying the reader details, the system needs to check whether the reader has any overdue loans, as required by the business constraints, which is done by the operation anyOverdueLoans().

In this way, the analyst will examine the interactions between the two components step by step, establishing the operations, parameters, return information (for clarity usually written in the operation list that follows the diagram rather than in the diagram) and control structure.

As can be seen in the protocol model, most of the early operations are about making various checks to ensure the business constraints are not breached. They can be called query operations. As indicated, major permanent changes to the state of the database tend to happen towards the end of the functionality unit, in this case, the operation BorrowItem(). This operation is more complex than may appear at first sight. It essentially signals to the system that a particular book has been borrowed by a particular reader. First, the system will have to determine the return date for the loan. Then it will have to record the loan. The system may then have to update the status of the book and so on. A lot of updating may happen here, which is also analysed in FEM in Section 11.5.

In this example, most operation calls originate from the boundary component, which are fired at the business component. This is because the interface of the application is perceived as user-driven. This is by no means universal. Indeed, there is a case for a server-driven operation call, i.e. an operation of the boundary component called by the business component. In this same example, after

some fine has been paid using payFine(amount), the control of execution is passed back to the point before the fine is retrieved using getFine(). This could be designed differently; the business component can be made to 'refresh' the amount of the fine displayed in the boundary component after some of the fine has been paid by the reader by calling an operation in BorrowBookUI. Component modularity requires that such mutual dependencies between components are, if not necessary, to be avoided. Diagrammatically, the server-driven operation calls can be marked using thick circles. The operation is added to the required service compartment of the business component and provided service compartment of the boundary component.

Only one protocol model is produced for Borrow Book, which need not be the case for every functionality unit. In fact, there can be a number of protocol models for a functionality unit, if it needs to be designed to allow different modes of interactions. For example, there are functionality units that may be used as both online interactive and offline bulk process. In such cases, the protocol between the boundary and business objects will be different, requiring two separate protocol diagrams for the single functionality unit. If there are multiple interface components, then the business component may also hold session information internally.

PM Step – 3 Produce the operation list from the protocol model. The analyst will get a list of operations that both the boundary component and the business component need to call. For these functionality units that have different protocol models, the list should consolidate the operations by removing repeated operations.

In the example, the component that offers Borrow Book service and the component that uses it need to offer the operations listed in Figure 12.5.

When defining the interface of a logical business component, analysts are likely to come up with a long list of operations, which can be rather daunting to manage. One way of dealing with this effectively is to divide the operations into groups by looking at them from higher levels of abstraction. LFD allows analysts to identify common elements among functionality units, and also express some of the activities in hierarchical manners. When working on USDM and protocol model, analysts should utilise this knowledge. For example, in Figure 12.5, the first eight operations, about ensuring that the reader is cleared to borrow a book, can collectively be called 'check reader clearance'. The same activity may be necessary for another functionality unit, say Reserve Book. In that case, all eight operations can be 'reused'.

At the end of this modelling the analysts would establish the interface between the two components in terms of operation signatures, and some dependencies among the logical components.

12.4 Operation List

A list of operations that the service providing component must offer and that the component requiring the service must use can be derived from the protocol diagram. This list provides an opportunity to elaborate the signatures of the operations identified and add documentational comments.

Figure 12.5 Operation List for Borrow Book

doesReaderExists(ReaderID): Status
getReaderDetails(): ReaderID, ReaderName, Address, TotLoans, TotRes, TotFine
anyOverdueLoans(): Status
getOverdueItems(): BookID, Title, Author, DateOut
getFine(): Fine
payFine(amount): ...
isFineLimitReached()
isMaxLoanNoLimitReached()
doesItemExists(BookID)
getItemDetails()
isItemReservedByOther()
BorrowItem()

12.5 Information Modelling

IM, discussed extensively in Chapter 11, can be used to model the structure of parameters passed in the operations identified in the protocol analysis. For example, the operation getReaderDetails() returns details of the reader, getOverdueItems() returns the loaned book details and so on. These parameters can be collected and analysed according to the technique describe in IM Steps 5 – 7 of Section 11.3 in order to ensure that entities have the right attributes.

12.6 Logical Component Specification

By this stage, analysts can produce precise specifications of logical components of the application in terms of the following:

- Description of the functionality unit – to verify the functionality of both boundary and business components
- LSL – to verify the visual layout of the boundary component
- USDM – to verify the interaction between the boundary component and the user
- Operation List – to verify the operations of the boundary and business components
- Protocol Model – to verify the interaction between boundary and business component
- IM fragment – to verify that the business component can access the shared database

12.7 Development Process

This modelling is necessary only when reuse of an entire application is not possible. In which case, the application is decomposed with the view to identify smaller-grained components for reuse, and possibly for development too.

12.8 Architecture

One of the main tasks of Application Manager is to define the interfaces of the components by registering the services and their operations with the Backbone component (see Section 6.2.2). It is worth remembering that the nature of physical components is yet to be determined; different components will offer and use different sets of services. If prefabricated components are to be deployed, the logical specifications of service components will suffice. When the physical components are

installed, the Backbone component will have to verify their interfaces against the service specifications of logical components, i.e. operations.

Chapter Thirteen

NAVITA System Modelling
Component Design

13.1 Introduction

Architecture analysis discussed in the previous chapter deals with the breakdown of the system into logical components and specification of their interfaces. Specification of an interface, in terms of its functionality, user interface, operation list, protocol model and data structure only goes so far as describing what a component providing a particular service unit should do. It is an external view of a component and does not suggest how the component should be implemented. If the developers assume that components need to be developed, concrete designs need to be produced. This chapter deals with issues surrounding the production of physical designs of those components. Since NAVITA distinguishes between business components and boundary components, design concepts, modelling process and techniques for these components will be discussed separately. Discussions in this chapter will concentrate largely on the business component modelling because reuse of business components is thought to be of more significance (Allen and Frost, 1998). Guidelines for boundary component design will spell out only the major steps, providing useful references for further discussions.

13.2 Physical Boundary Component Design

A physical boundary component implements one or more logical boundary component(s). Apart from usability issues such as user-friendliness of the components, the main issue here is to package together interface objects that are shared by related functionality units.

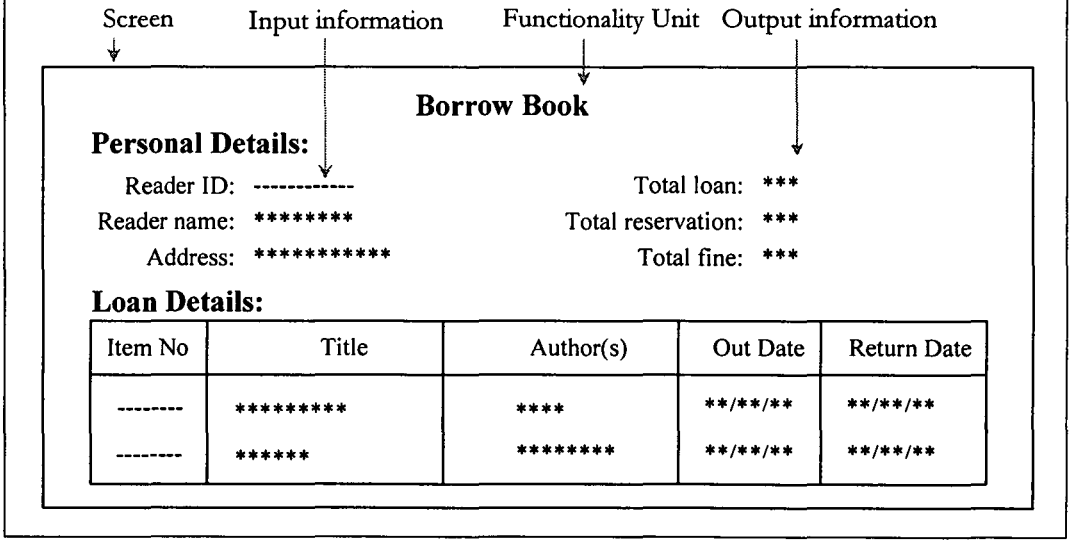
13.3 Physical Boundary Component Design: Modelling Concepts

The main concepts used in this model are boundary classes and their relationships. Although the concepts of class and class relationships described in Chapter 11 are mainly used for modelling business classes, these concepts can also be applied to boundary classes too.

13.4 Physical Boundary Component Design: Modelling Process and Technique

Chapter 10 provides discussions on how to produce logical screen layouts containing various visual objects that users use to interact with the system. If these objects are to be implemented, the following steps should be taken:

Figure 13.1 Logical Screen Layout for Borrow Book



BoCD Step – 1 Confirm all LSLs and USDMs with the users.

BoCD Step – 2 Identify common visual elements used in functionality units.

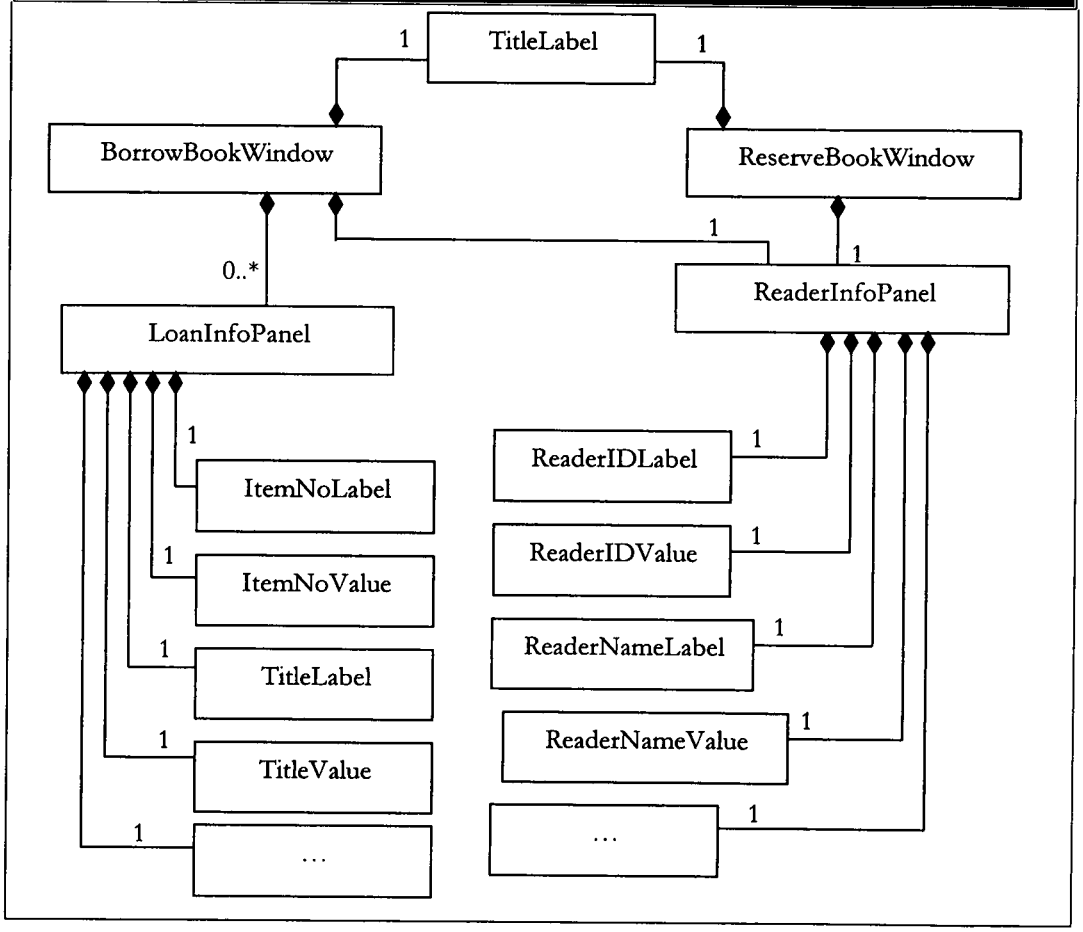
In Figure 13.1 for example, reader details such as Reader ID, Reader name, Address, Total loan, Total reservation, and Total fine could also appear in other FUs, such as 'Reserve Book'.

BoCD Step – 3 Produce a boundary class diagram for the objects used in the boundary components, including the common elements. If there

are a large number of visual objects in the component, packaging mechanisms such as UML Package Diagram (OMG, 2003) can be used.

Figure 13.2 shows a partial boundary class diagram indicating how the GUI objects of Borrow Book can be composed and how some of these objects can be shared with other functionality units such as Reserve Book.

Figure 13.2 Partial Boundary Class Diagram



BoCD Step – 4 Decide the implementation technology, such as Java, Visual Basic.

BoCD Step – 5 Revise the design by adding libraries. Design patterns (Gamma et al, 1995) can also be applied.

BoCD Step – 6 List operations in component diagram.

13.4.1 Related Work

There is a wealth of material on UID, for which Shneiderman and Plaisant (2004) and van Harmelen (2001) provide a good starting point.

13.5 Physical Business Component Design

Section 6.4 discusses the nature of physical components, and Chapter 12 discusses the models, tools and techniques for producing specifications of logical components. Since logical specifications are for each functionality unit, implementation of a service by a component is not realistic in most situations. Not only is it technically more difficult to deal with a very large number of components, there are also many other issues such as integrating a large number of dependent components into an application and reusing small-grained components. Therefore, one of the key questions in component design is concerned with the composition of logical services into cohesive physical components.

Component design in NAVITA makes use of the following diagrams:

- Component Diagram
- Sequence Diagram
- State Transition Diagram

13.6 Physical Business Component Design: Modelling Concepts

The component diagram is used to show statically how physical components make up an application. The main modelling concepts used in this diagram are:

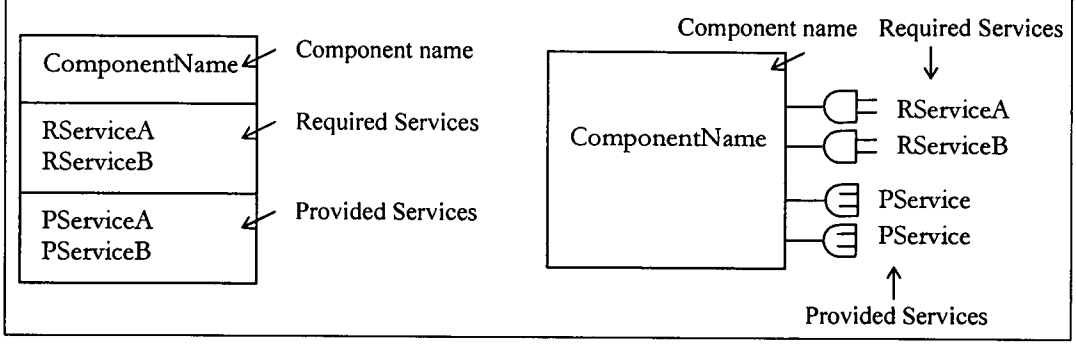
- Physical Component (Boundary, Business and other components)
- Backbone Component (Section 6.2.1)
- Application Administrator (Section 6.2.3)

13.6.1 Physical component

A physical business component is an implementation of a service, or a set of related services, defined by logical service specifications. In this diagram, logical services are translated into physical components in such a way that they address both business and technical concerns. NAVITA physical components can be notationally

represented by two different conventions. In one convention, the component name is written in the first of the three compartments, while the other two compartments list services the component provides and requires. In the other convention, services are represented by plug-socket icons, as shown in Figure 13.3.

Figure 13.3 Physical Component



13.7 Business Component Physical Design: Modelling Process and Technique

As explained, one of the main tasks in this modelling is to translate logical services into physical business components by grouping related services together. Section 6.4 suggests that there are two main existing approaches to composing components. The solution also discusses their flaws and proposes the NAVITA solution to these problems. According to the discussion, NAVITA components must reflect both business and technical perspectives. It should be emphasised that if components are defined only from the way their services are used, i.e. the business perspective, they will not provide a stable basis for component composition because all services are related in some way, and it is difficult to determine which services should and should not be included. On the other hand, the technical perspective does provide a stable basis for component composition, but if the component has no relevance to the way in which services are used, its reusability is limited. The main issue is to find the point at which the two perspectives can converge. NAVITA suggests that FEM holds the key to this. The FEM in Figure 13.4 shows all the major functionality units and entity classes in the library system together with the effects of functionality units.

Figure 13.4 FEM for LibrInfoSys

	Reader	Reservation	Loan	Title	Copy	Author/Title	Author
Add Book				I L_1 L_3	I L_1	I L_2 L_3	I L_2
Add Copy				L	I L		
Register Reader	I						
Search Catalogue				R*	R*	R*	R*
Borrow Book	R L_1 M	[R M]	IL_1 L_2	R	R L_2	R	R
Return Book	R C_1		R M C_1 C_2	R	R C_2	R	R
Renew Loan	R		R M	R	R	R	R
Reserve Book	R L_1	IL_1 L_2		RL_2	R	R	R
Cancel Reservation	R C_1	C_1 C_2 D		RC_2	R	R	R
Show Reader's current loans	R		R*	R*	R*	R*	R*
Send Reminder	R		M	R	R	R	R
Update Reader Details	R M						
De-register Reader	R C_1 C_2 D	C_1	C_2				
Remove Book				R C_1 C_3 D	R C_1 D	R C_2 C_3 D	R C_2 D
Remove Copy				R C	R C D	R	R

Keys:

I = Insert M = Modify D = Delete R = Read

L = Link C = Cut a link * = Multiple effects [] = Optional Effects

BuCD Step – 1 Rearrange the ordering of the functionality units in the FEM on the basis of the effects they have on entities. Group together functionality units which affect the same entity classes.

At first glance, correlations between the services/functionality units and entity classes in the FEM for the library system may appear somewhat random, i.e. the way in which services can be grouped, will not have much bearing on the way entity classes can be grouped. In fact, the correlations become evident when the matrix is rearranged by listing FUs that affect the same entities together (see the order of functionality units in Figure 13.5). It becomes apparent that certain sets of functionality units affect a similar set of entity classes and these sets of functionality units also tend to be related in the business sense. For example, book operations.

BuCD Step – 2 Examine the nature of relationships between the entity classes. As discussed in various subsections of Section 6.4.2, certain entities simply elaborate the data structure of the objects they are representing, and do not constitute substantial classes in their own right. Each group of these classes is then listed as a single class, and effects on the lower-grained classes are combined. Effects that are entirely internal to the new classes should be hidden.

Title, Copy, Author/Title and Author in Figure 13.4 can be represented by a single class, such as Book. Furthermore, Loan and Reservation define complex relationships between Reader and Book, known as link entities. These entities can be either grouped with Reader or Book; in this case, the latter is preferable because the entities define the complex data structure of links books have to readers. Therefore, there are two main components in the system as shown in Figure 13.5.

As far as the effects are concerned, Add Book, for example, has many pairs of Ls in Figure 13.4 which are now completely hidden

inside Book; therefore, the external view of Book for Add Book only shows that the functionality unit is about adding an instance of the class. The process of creating instances of multiple smaller classes and linking them is now hidden. However, if one half of a pair of effects affects another entity, then the effect needs to be made externally visible, as for Borrow Book.

Figure 13.5 Compacted FEM

	Reader	Book
Register Reader	I	
Update Reader Details	R M	
Add Book		I
Add Copy		I
Remove Book		RD
Remove Copy		RD
Search Catalogue		R*
Reserve Book	R L_1	IL_1 R
Cancel Reservation	R C_1	C_1 D R
Borrow Book	R L_1 M	[M] IL_1 R
Renew Loan	R	R M
Send Reminder	R	M R
Return Book	R C_1	R M C_1
De-register Reader	R C_1 C_2 D	C_1 C_2
Show Reader's current loan	R	R*

Keys:

I = Insert M = Modify D = Delete R = Read

L = Link C = Cut a link * = Multiple effects [] = Optional Effects

It is now clear from Figure 13.5 that related FUs tend to have effects on similar sets of entity classes. There are three main

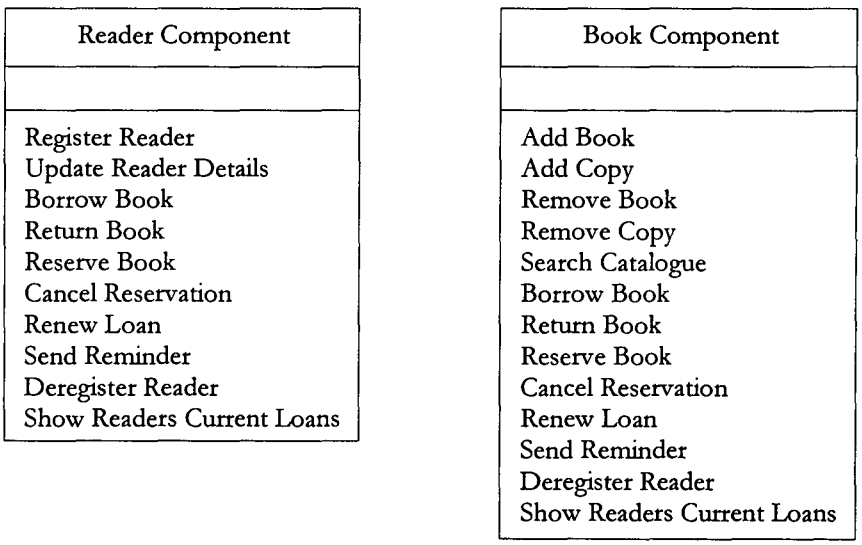
groups of related FUs, dealing with maintaining reader information, book information and information about readers' activities respectively. The first two groups affect two main entities, Reader and Book, whilst the last group affects both.

FUs in these groups are not only logically related, in terms of the way they can be used by the users, but also in agreement with the underlying structure of the system.

BuCD Step – 3 Draw the first sketch component diagram based on the entity 'classes' and services. Represent each class as a component.

The revised entity class diagram now includes only two significant 'classes' in the system, Reader and Book, and two relationships between them, Reservation and Loan, which are largely encapsulated by Book. These classes form the basis for NAVITA components because they encapsulate tightly coupled entity classes and also provide a set of related business services.

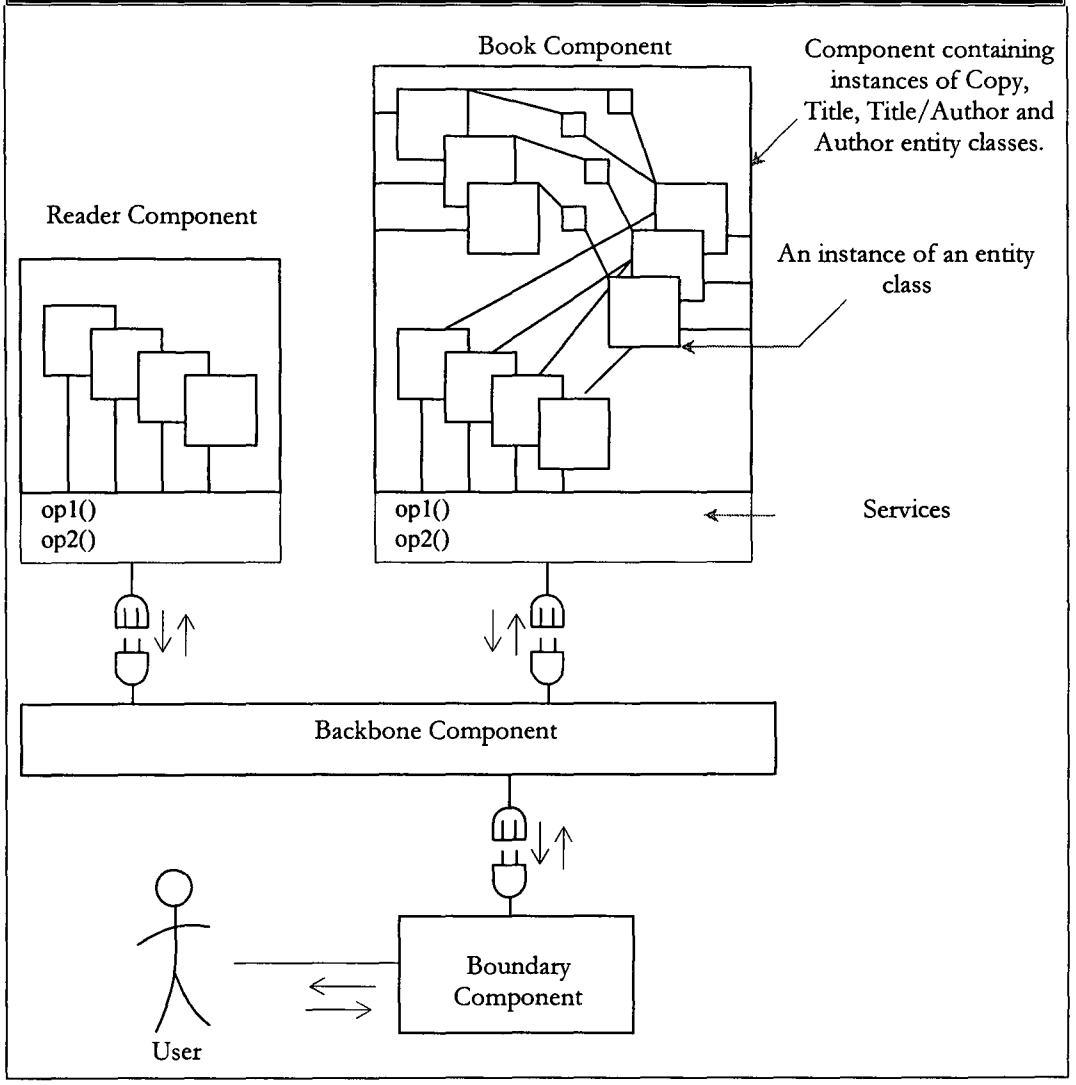
Figure 13.6 Component Diagram for LibrInfoSys



BuCD Step – 4 List in the bottom compartment of a component all services the component provides without collaborating with another component. For FUs that affect multiple components, services are

listed in all affected components.

Figure 13.7 Components and Entity Objects at Run Time



In this example, Add Book and Register Reader only affect individual components, while FUs, such as Borrow Book, affect both Reader and Book components. Therefore, they are listed in both components. In these cases, the same service names should be given to components, but each component will have a different effect. For example, when a book is borrowed, from the Reader component point of view, the status of the appropriate reader needs to be updated. From the point of view of the Book component, the status of the appropriate book needs to be updated. As far as the boundary component is concerned, it stills

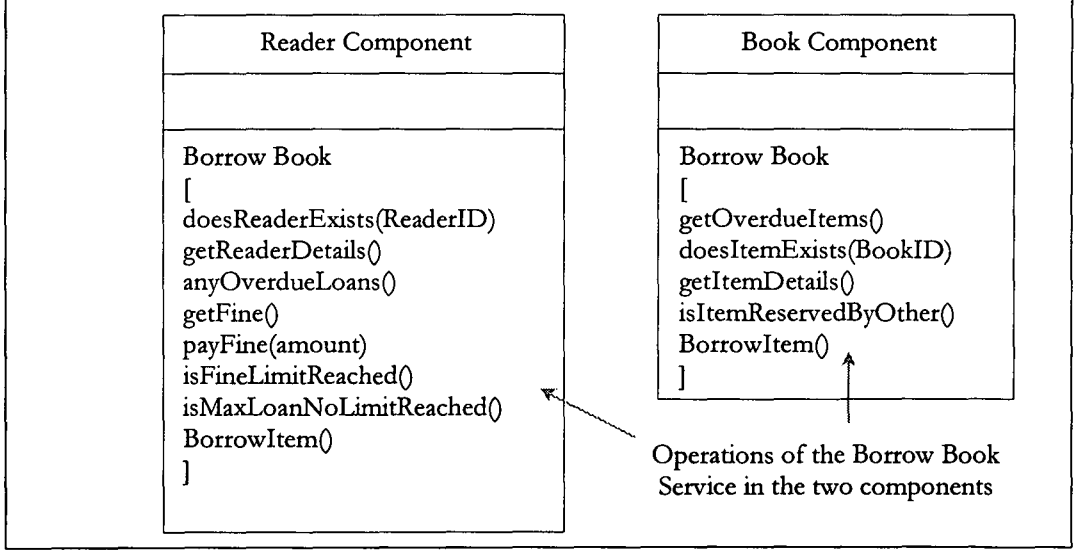
carries on communicating with the business components as if they were one.

It is worth noting that the components shown in the diagram are not classes in the traditional sense. These components also act as containers that hold instances of classes, record types and so on. Therefore, the container itself has responsibilities, such as maintaining session information.

BuCD Step – 5 For FUs that affect multiple components, the operations identified in the protocol model are split and allocated to appropriate components. Allocation is based on the information the operations access (See 13.8.1).

For example, Figure 12.5 shows the list of operations for Borrow Book. Based on the discussion in Section 13.8.1, operations such as doesReaderExist(ReaderID) and getReaderDetail() are to be allocated to the Reader component, while operations such as doesItemExists(BookID), are to be provided by the Book component. The division of the protocol model requires revising of the session structure of each component; each component now has its own internal session.

Figure 13.8 Splitting Operations Across Components



BuCD Step – 6 Convert relationships between the entity classes into constraints of individual components. Relationships between components are therefore implemented locally.

When a book is borrowed, the reader object keeps a reference to the appropriate book object, and vice versa. Each component reinforces its own constraints. For example, when a book is borrowed, the book component ensures that another reader has not already reserved it whereas the reader component ensures that the reader has not exceeded the loan limit. (See Section 13.8 for discussion on rational allocation of operations.)

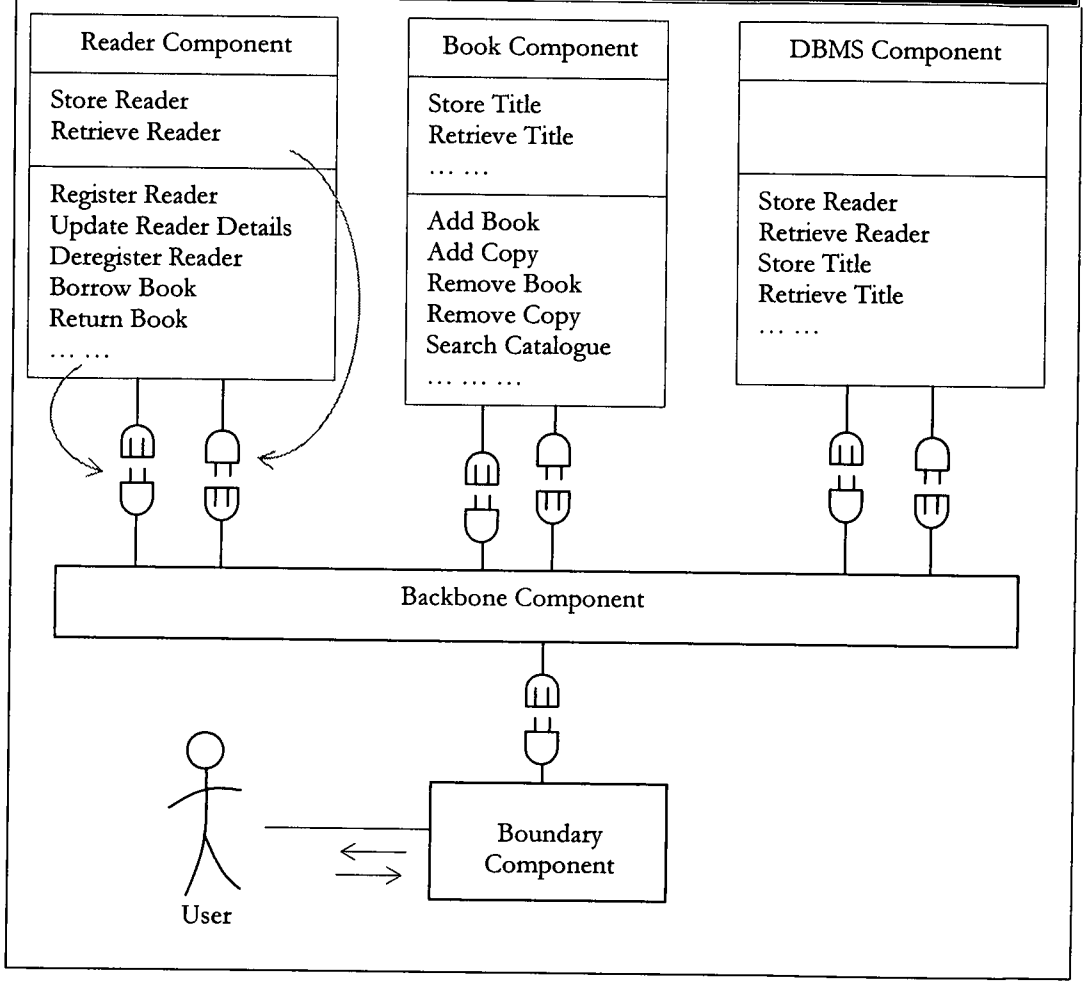
Another key issue here is ensuring consistency of updates. This means that if a functionality unit affects two components, the two updates must be kept in sync with each other. Since, the effects are atomic, the precedence of updates does not matter. For example, for Borrow Book, it does not matter whether Book or Reader components are updated first. What matters is that both or neither are updated. Therefore, when the boundary component asks to complete Borrow Book, the Backbone component calls the BorrowItem() operation. As both Reader and Book components have the operation BorrowItem() (Figure 13.8), the Backbone component broadcasts the operation call, so that both components update their own states. At the end of the service, each component destroys its own session.

BuCD Step – 7 Determine the implementation technology. Then revise the component diagram by adding implementation-specific components.

In addition to the business services, business components may also require other services, in particular data storage services that hold permanent information in the system. Data services will only provide basic operations such as saving and retrieving an object or a record. Therefore, they could be implemented using simple

text files. If a more advanced DBMS is to be used, all the services that come with the DBMS will also be available. Note that access to DBMS operations go through the Backbone component. That is, all components talk to the same Backbone component in order to access services of any kind. Services are added to the middle compartment of each business component specification to indicate the DBMS services required.

Figure 13.9 Physical Architectural Model of LibrInfoSys



13.8 OO Design for Business Components

Since physical architecture or component design is technology-biased, it is necessary to determine the implementation technology(s) for this design. If a more traditional development technology, such as structured programming languages and a relational database, is to be used for implementing the business components, service

operations can be translated into program modules, and entity classes and their relationships into relational databases. However, implementation technologies that are widely used nowadays, such as Java, are mainly OO. Although Java is not a component-based programming language, it is well recognised for providing a range of mechanisms to help create components. For this reason, business components may need to be translated into an OO-based design. If the components are to be implemented using OO technology, certain adjustments have to be made, mainly by allocating operations to entity classes and applying inheritance.

For each business component, analysts now have their service specifications, entities, and the protocol model. If these are to be converted into an OO design, the first thing analysts have to do is to allocate operations to entity classes.

13.8.1 Principle on distribution of operations

One of the main activities of this research is an investigation into the basis on which allocation of operations to classes can be determined. Since this is an important issue that has relevance to OO methods, a detailed investigation has been carried out. Findings of the investigation are formatted as a journal paper and attached to the Appendix III. The paper identifies weaknesses in existing OO methods regarding the issue and proposes a set of two principles, which provide a rigorous foundation for rational allocation of operations to OO classes.

The first principle deals with the basic criteria that need to be met in order to justify allocation of an operation to a class. The principle states:

> An operation allocated to an object must access the properties of the object in order to justify the allocation. It therefore means that at least one of the following conditions needs to be met.
>
> Criterion 1: The operation accesses the concrete attribute(s) of the object.
>
> Criterion 2: The operation accesses the derived attribute(s) of the object.
>
> Criterion 3: The operation accesses the states of link(s), i.e. existence or non-existence of links, the object has to other objects.

Building on the first principle, the second principle states:

> Operations of classes have achieved fair distribution of operations if all of the following criteria are satisfied ubiquitously.
>
> Criterion 1: The operation allocated to an object does not defer what can be done in the object to another object. What can or cannot be done by an operation of an object or how much an operation can do is ascertained from the properties of the object. The amount of work an operation performs must not transcend the properties of the object.
>
> Criterion 2: Every operation call between two objects follows the static associative relationship that exists between the class or classes of the two objects.
>
> Criterion 3: Control – the task of calling operations – is distributed in such a way that it reflects the chains of properties that exist among objects. For each chain of properties, there has to be an object that serves as the starting point of the chain. If there is none, creation of a control object is necessary.

These principles are observed in NAVITA OO component design, which uses two UML diagrams: Sequence and State Transition Diagram. The UML Collaboration Diagram can be deployed instead of the Sequence Diagram; however, NAVITA recommends the latter.

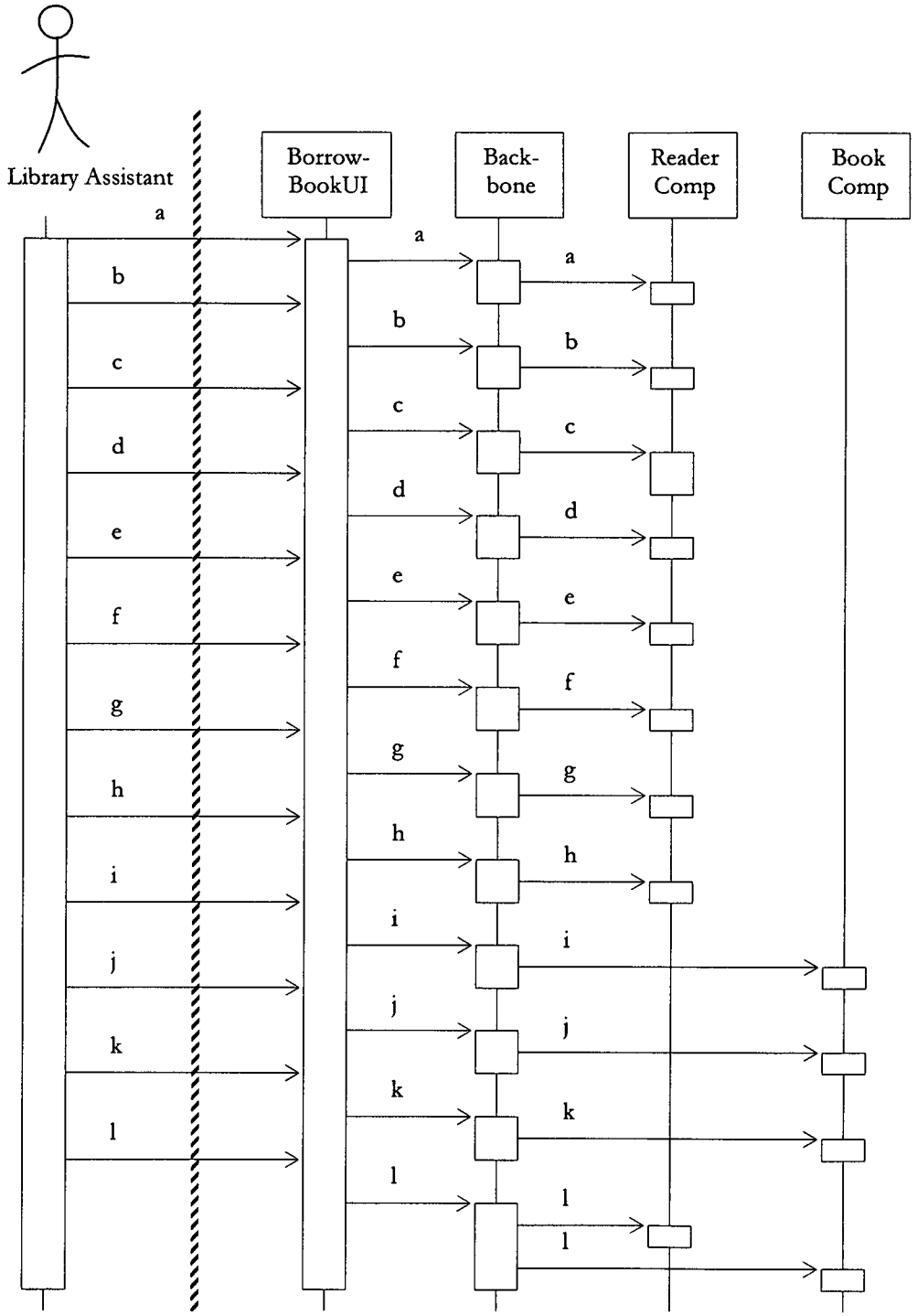
13.8.2 Sequence Diagram

This diagram can be used for a number of purposes, such as showing how physical components communicate realising a specific functionality unit, and how objects within a component will communicate to realise a specific service.

The main modelling concepts are:

- Components/Objects that participate in the collaboration
- Messages passed between them
- The order of the messages passed

Figure 13.10 Initial Sequence Diagram for Borrow Book functionality unit



a = doesReaderExists(ReaderID)
b = getReaderDetails()
c = anyOverdueLoans()
d = getOverdueItems()
e = getFine()
f = payFine(amount)
g = isFineLimitReached()
h = isMaxLoanNoLimitReached()
i = doesItemExists(BookID)
j = getItemDetails()
k = isItemReservedByOther()
l = BorrowItem()

The main inputs to this modelling are Service Specification, FEM, and the class distribution principles. For each functionality unit, a service specification shows the operations, entity classes and protocol model.

SeqD Step – 1 For collaborations between components, candidates for participating components are identified from the 'componentised' FEM, the FEM with NAVITA components. For collaboration of objects within a component, candidate classes are identified from the earlier FEM. Other sources, such as descriptions of the functionality unit and the IM fragment produced for the functionality unit, may also be useful.

Figure 13.10 shows components that are involved in the realisation of the functionality unit Borrow Book. Participation of the Reader and Book components in this functionality unit can be seen in the FEM in Figure 13.5. The sequence diagram in Figure 13.12 shows how objects with the Book component interact to realise the Add Book functionality unit. Participating objects in the diagram are identified from the FEM in Figure 13.4.

SeqD Step – 2 Operations are taken from the operation list and step-by-step description of the functionality unit. Effects in FEM can be analysed for further identification of operations. Draw a sequence diagram based on the initial allocation of operations.

Figure 13.10 shows interactions between the actor, boundary component, Backbone component and business components for the functionality unit Borrow Book. Operations identified in the protocol analysis in Figure 12.5, are allocated to the two business components.

Operations of collaborating objects in Figure 13.12 can be derived by studying the effects on the classes in FEM in Figure 13.4. Title, for example, has one I and two Ls, indicating that a new instance of the class needs to be created and linked to Copy and Title/Author. Operations are allocated to appropriate classes

to implement those effects.

SeqD Step – 3 By applying the class operation principles, operations are allocated to entity classes and the control distributed. Revise the sequence diagram.

The allocation of operations in Figure 13.10 is based on the principles referred to in Section 13.8.1. For example, the operation doesReaderExists(ReaderID) is clearly something a Reader component can do: the component, like a control object discussed in the operation allocation principles, holds references to all objects inside. By going through the list of references to the reader objects, the component can determine whether details of such a reader exist. Once the object is identified, details of the reader can be retrieved for the next operation. getOverdueItems() is more complex; it may appear that the operation should have been allocated to the Book component with the reader ID as a parameter to retrieve details of any overdue loan. However, the class allocation principles suggest that, since all the interested loans are linked to a reader object, that object is the starting point for this query. Reader object only holds references to loan objects in the Book component. Therefore, through the Backbone component, the reader object will ask the book component whether any of the loans it refers to are overdue, and if so, obtain the book details, as shown in Figure 13.11.

In Figure 13.12, the container of the Book component acts as the control object, as required by the class allocation principles, and it creates various instances of the classes.

SeqD Step – 5 The component diagram is revised by adding the operations to components and its entity classes.

From Figure 13.11, it is clear the operations IsLoanOverdue() and getLoanDetails() need to be provided by the Book component as part of the Borrow Book service. Figure 13.12 indicates that

various constructor and link operations need to be added to the classes in the Book component.

Figure 13.11 Revised Sequence Diagram for Borrow Book Functionality Unit

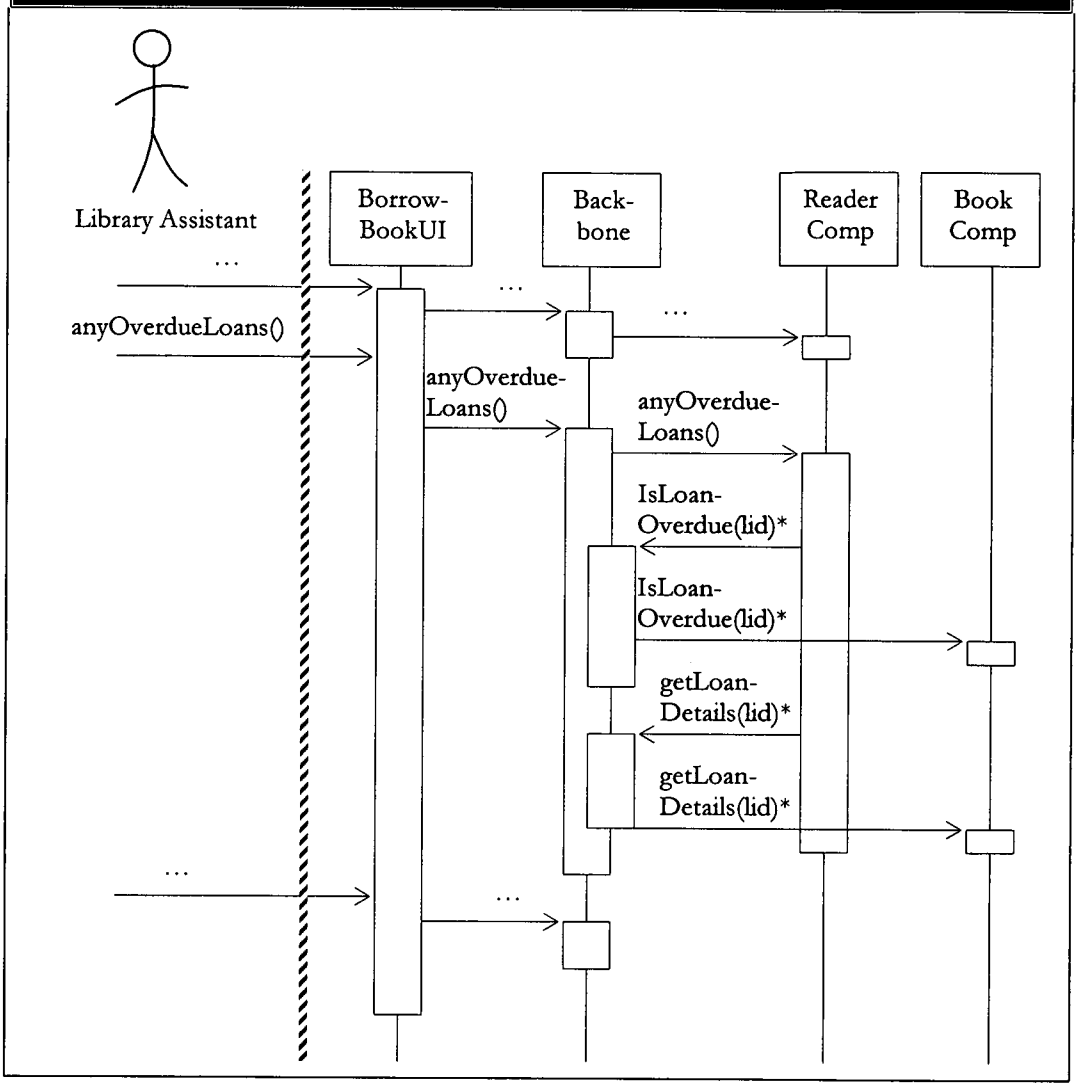
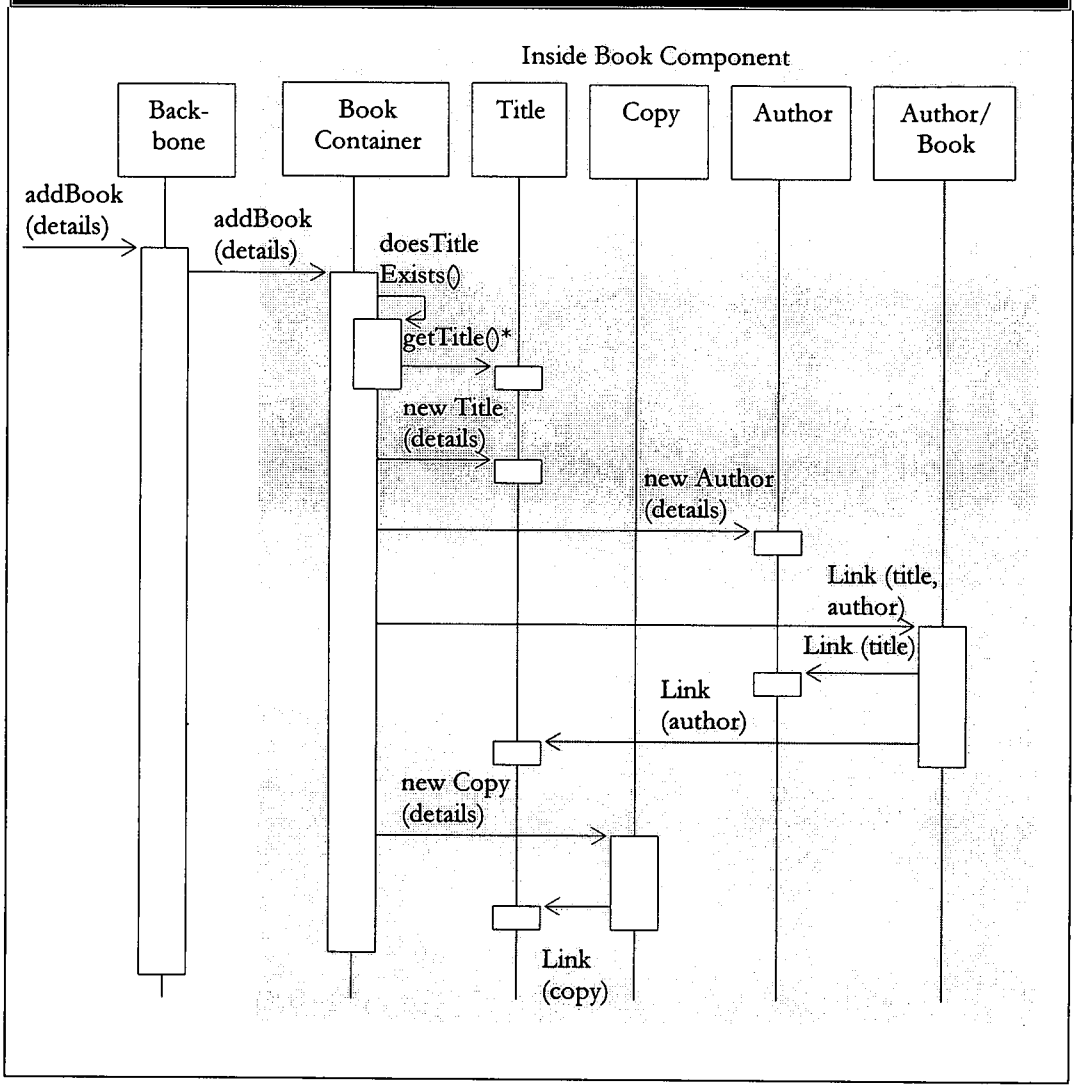


Figure 13.12 Partial Sequence Diagram showing interactions between objects of the Book component for the Add Book functionality unit



13.9 State Transition Diagram

NAVITA State Transition Diagram (STD) is used to show the state changes in the lifetime of an object or a component. This diagram is largely based on the UML Statechart Diagram (OMG, 2003).

13.10 State Transition Diagram: Modelling Concepts

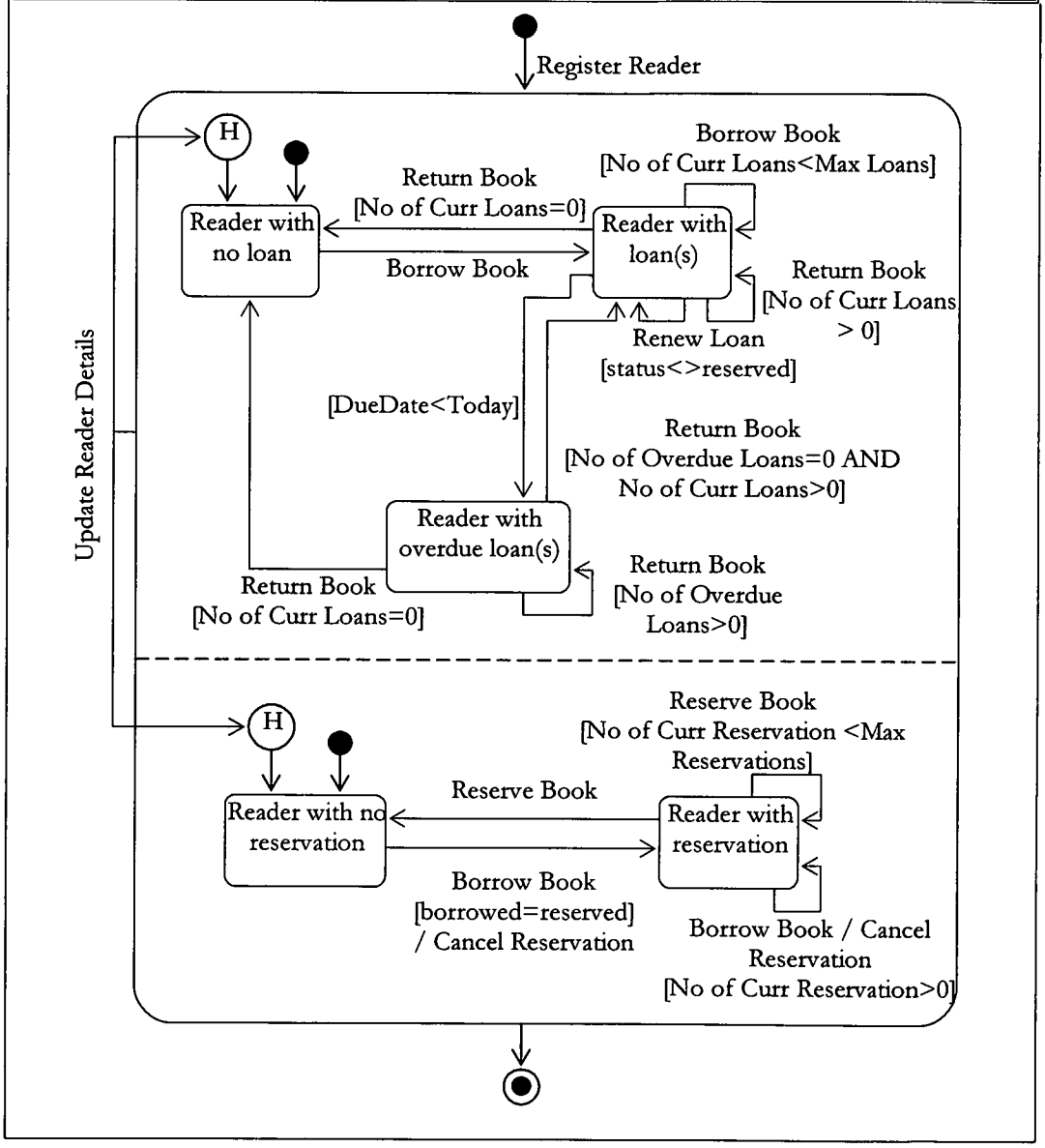
The main concepts used are:

- Object/Component (Section 11.2.1 and 6.5)
- Event and Transition
- State

13.10.1 Event and Transition

In NAVITA, events are identified from the FEM, in which each functionality unit with an update effect on the component or object is regarded as an event. The correspondence is always one-to-one because each functionality unit is atomic. Transitions of states caused by these events are explored using a state table. Events and transitions are represented by arrows, as in UML.

Figure 13.13 State Transition Diagram for Reader Component



13.10.2 State

State is the condition of an object or a component, which affects its behaviour. States of an object are determined by its attribute values and the links the object has with other objects. States of a NAVITA component are typically determined by states of a set of objects encapsulated by the component. For example, states of the Book component are determined by the collective states of a set of Title, Copy, Title/Author, Author, Loan and Reservation objects representing a single book. Refer to UML Specification (OMG, 2003) for a more in-depth discussion on the concepts used in this model, such as sub-states, concurrent states, guard condition, and so on.

13.11 State Transition Diagram: Modelling Process and Technique

STD Step – 1 Identify the FUs that affect the state of the component/object for which STD needs to be drawn. FUs with only R effects do not change the state of the component/object.

For example, Reader is affected by Register Reader, Borrow Book etc, not Show Reader's Current Loans.

STD Step – 2 Determine the possible states of the component/object before and after each functionality unit by constructing a simple state table.

Table 13-1 shows the states of the Reader component before and after being affected by FUs.

STD Step – 3 Draw a STD showing the complete lifecycle of the object or component.

Figure 13.13 shows STD for the Reader component.

Table 13-1 States of Reader Component Before and After some FUs			
	"Before" State	"After" State	Repetition
Register Reader	Null	Reader with No Loan and Reservation	None
Update Reader Details	Not null	Same as Before State	Many
Reserve Book	Reader with no loan Reader with loan Reader with reservation Reader with no reservation	Reader with reservation	Up to reservation limit
Cancel Reservation	Reader with reservation	Reader with reservation/ Reader with no reservation	Up to number of current reservation
Borrow Book	Reader with no loan Reader with loan Reader with reservation Reader with no reservation	Reader with loan	Up to loan limit,
Renew Loan	Reader with loan	Reader with loan	Up to renewal no limit
Send Reminder	Reader with overdue loan	Reader with overdue loan	Up to reminder no limit
Return Book	Reader with Loan/ Reader with overdue loan	Reader with loan/ Reader with overdue loan/ Reader with no loan	Up to no of current loan
De-register Reader	Reader with no Loan and Reservation	Null	Once

13.12SDP

Modelling of physical boundary components and physical business components occurs in Stage 8 – Physical Design.

13.13Architecture

Modelling of physical components is all about physical architecture. NAVITA models help analysts carry out a systematic analysis of physical business components, addressing both business and technical dimensions of reusable components.

Chapter Fourteen

Evaluation of NAVITA

14.1 Introduction

This chapter evaluates the new CBSD method NAVITA by applying the same rigour and criteria of the MAP framework as used in the evaluation of the existing CBSD methods. Evaluation results are then compared.

Evaluation of each existing CBSD method in Chapter 4 is preceded by a summary of the method highlighting its key features. Since NAVITA has been described extensively in previous chapters, it need not be described here again.

14.2 Evaluation of NAVITA

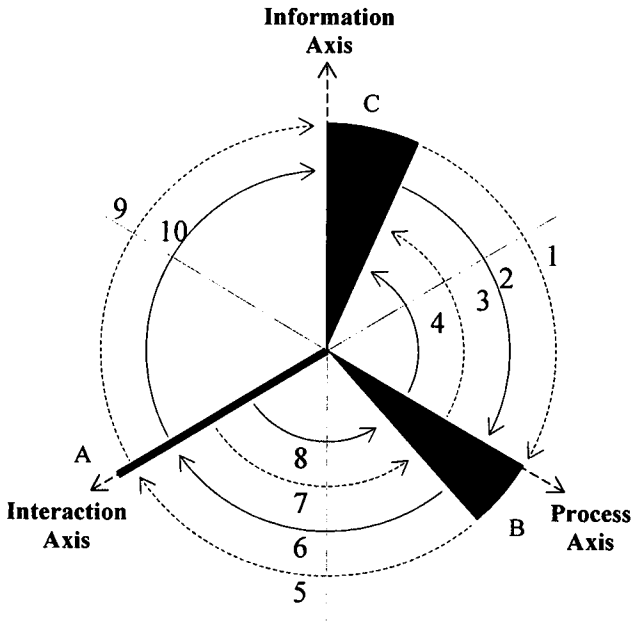
NAVITA provides all three elements of a method as required by the MAP framework. The following correlations between the three elements of a method can be noted.

- Correlations between modelling and SDP stages are made explicit at the end of each chapter on modelling; for example, Section 8.6.
- The relationships between the system development process and software architecture are made clear through development steps such as Step 6 and Step 7, see Sections 7.2.6 and 7.2.7
- The correlations between the modelling and software architecture are made clear at the end of each chapter on modelling; for example Sections 8.6 and 8.7.

Therefore, there is a high level of interconnectedness between elements of NAVITA.

14.2.1 Evaluation of System Modelling

Figure 14.1 IPI Matrix for NAVITA Models



A. Context Diagram
B. Middle-level Functionality Diagram (MFD)
C. Information Model (IM)
1. A column in Functionality unit Entity class Matrix (FEM)
2. State Transition Diagram
3. A row in FEM / Logical Architectural Analysis
4. Sequence Diagram
5. A middle-level functionality unit involves an event-level interaction
6. User System Dialogue Model / Lower-level Functionality Diagram (LFM)
7. Implicit in MFM
8. LFM
9. Logical Screen Layout
10. Protocol analysis (IM fragment and Operation List)

The IPI Matrix in Figure 14.1 shows NAVITA models. The following conclusions can be drawn about the coverage of these models:

- The method provides all three global models as required by the MAP framework.
- There is a high number of abstract contextual models: five out of six.
- The coverage of detailed contextual models in NAVITA is also extensive. There are no abstract and detailed contextual models in NAVITA that show how an entity/class relates to one or many interactions. This is largely because there is a general one-to-one relationship between middle-level

processes and interactions – with contextual diagrams showing how one entity/class is affected by processes (1 and 2) – other diagrams in this case will be extraneous.

14.2.1.1 Evaluation of Context Diagram

Table 14-1 summarises evaluation of the modelling technique for the diagram. Modelling technique for this global model is fairly rigorous and many contextual models make cross-references to and from this diagram.

Table 14-1 Evaluation of the NAVITA Context Diagram Modelling Technique

Criterion	Guidelines	Rigour
(a) Completeness of system boundaries	CD Step 1: Two system boundaries are added first, one of which may be removed later.	2
(b) Completeness of actors	CD Steps 2-4: Identification, careful categorisation, positioning of actors according to their roles, together with the users' involvement in the process ensure this completeness.	2
(c) Completeness interaction	CD Step 5: A general one-to-one mapping between interactions and functionality units, subsequence checks with MFD, LFD, LSL and USDM ensure completeness and correctness of interactions.	2
(d) Minimality of system boundaries	CD Step 6: In specific cases, the greater IS boundary is removed.	2
(e) Minimality of actors	CD Step 7: Actors and interactions are eliminated according to the changes in the system's boundaries.	2
(f) Minimality of interaction	CD Step 7.	2
(g) Correctness of system boundaries	Implicit in CD Step 7.	1
(h) Correctness of actors	CD Step 2: User participation in the process ensures that actors are appropriately represented.	2
(i) Correctness of interaction	CD Step 5.	2
(j) Non-redundancy of system boundaries	Not applicable.	0
(k) Non-redundancy of actors	CD Step 8: Consolidation of actor roles removes redundant actors.	2
(l) Non-redundancy of interaction	CD Step 9: Redundant elements in input/output data items are removed.	1
	Total	20

	Strength of the NAVITA context diagram modelling technique	91

14.2.1.2 Evaluation of Middle-level Functionality Modelling

Since this model and the previous model have many elements in common, evaluations of common elements are replicated from the previous table. Table 14-2 gives a summary of the evaluation of the MFD modelling technique.

Table 14-2 Evaluation of the NAVITA MFD Modelling Technique

Criterion	Guidelines	Rigour
(a) Completeness of functionality units	MFD Step 1: FUs are identified from a number of sources: business processes, prototyping, RAD etc. They are also cross-checked with the context diagram and. Furthermore in FEM Step 3, FUs are cross-checked with entity classes.	2
(b) Completeness of actors	See (b) in Table 14-1.	2
(c) Completeness interaction	See (c) in Table 14-1.	2
(d) Minimality of functionality units	MFD Step 2 and FEM Step 3.	2
(e) Minimality of actors	See (e) in Table 14-1.	2
(f) Minimality of interaction	See (f) in Table 14-1.	2
(g) Correctness of functionality units	Each MFD is further analysed by LSL and USDM. Users' involvement in the development of LSL and UDM can help validate the functionality units.	2
(h) Correctness of actors	See (h) in Table 14-1.	2
(i) Correctness of interaction	See (i) in Table 14-1.	2
(j) Non-redundancy of functionality units	MFD limits itself to 'complete' processes and there is no explicit analysis of, say, common elements between functionality units.	1
(k) Non-redundancy of actors	See (k) in Table 14-1.	2
(l) Non-redundancy of interaction	See (l) in Table 14-1.	1
	Total	22
	Strength of the NAVITA MFD modelling technique	92

14.2.1.3 Evaluation of Information Modelling

Table 14-3 summarises the evaluation results for this modelling technique.

Table 14-3 Evaluation of the NAVITA IM Modelling Technique

Criterion	Guidelines	Rigour
(a) Completeness of classes	IM Step 1 and IM Steps 4-7. Classes, attributes and relationships are identified from two lines of analysis – top-down grammatical analysis of requirements and bottom-up relational data analysis of LSL – before merging together.	2
(b) Completeness of attributes	IM Step 1 and Steps 4-7.	2
(c) Completeness of associative relationships	IM Step 2 and IM Steps 4-7.	2
(d) Completeness of inheritance relationships	IM Step 3. Inheritance relationships are identified, however, their completeness cannot be guaranteed.	1
(e) Minimality of classes	IM Steps 4-7. When class diagrams developed separately are merged together, superfluous classes, attributes and relationships are eliminated.	2
(f) Minimality of attributes	IM Steps 4-7.	2
(g) Minimality of inheritance relationships	Implicit from IM Step 3.	0
(h) Minimality of associative relationships	IM Steps 4-7.	2
(i) Correctness of classes	Attributes gather in IM Step 4 are based on User Interface design that are developed in collaboration with the users. Users are therefore able to confirm the attributes that they believe are necessary for their business operations. Entity classes are further based on these attributes. Therefore these entity classes have a high chance of being correct.	1
(j) Correctness of attributes	See (i)	1
(k) Correctness of inheritance relationships	Not provided.	0
(l) Correctness of associative relationships	See (i). Class allocation principles in Sequence Diagram modelling require that class/classes of communicating objects has/have associative relationships.	1
(m) Non-redundancy of classes	RDA in IM Step 5 requires removal of entities with similar sets of attributes.	2
(n) Non-redundancy of attributes	RDA in IM Step 5 ensures that derivable attributes are removed.	2
(o) Non-redundancy of	Not provided.	0

inheritance relationships		
(p) Non-redundancy of associative relationships	IM Step 5.	2
	Total	22
	Strength of the NAVITA IM modelling technique	69

14.2.1.4 Evaluation of USDM

Table 14-4 gives a summary of the evaluation of the USDM modelling technique.

Table 14-4 Evaluation of the NAVITA FEM Modelling Technique

Criterion	Guidelines	Rigour
(a) Completeness of I/O fields	USDM Step 1: I/O fields are taken from the LSL which has been validated by the user.	2
(b) Completeness of flows	USDM Step 2: Identified from the LSL, then crosschecked with functionality unit's description, LFD.	2
(d) Minimality of I/O fields	See (a).	2
(f) Minimality of flows	See (b).	2
	Total	8
	Strength of the NAVITA FEM modelling technique	100

14.2.1.5 Evaluation of Protocol Analysis

Table 14-5 gives a summary of the evaluation of the NAVITA protocol analysis modelling technique.

Table 14-5 Evaluation of the NAVITA Protocol Analysis Modelling Technique

Criterion	Guidelines	Rigour
(a) Completeness operations	PM Step 1 – IM Step 2: Operations are identified from the various sources, in particular from USDM. Completeness however cannot be ensured.	2
(b) Completeness of flows	PM Step 1 – IM Step 2: Identified from the USDM.	2
(d) Minimality of operations	See (a).	2
(f) Minimality of flows	Not provided.	0
	Total	6
	Strength of the NAVITA FEM modelling technique	75

14.2.1.6 Evaluation of Sequence Diagram

Table 14-6 gives a summary of the evaluation of the NAVITA Sequence Diagram modelling technuqe.

Table 14-6 Evaluation of NAVITA Sequence Diagram Modelling Technique

Criterion	Modelling Guidelines	Rigour
(a) Completeness of participating objects	SeqD Step 1. Affected objects/components are identified from FEM.	2
(b) Completeness of operations	SeqD Step 2. Operations are determined from descriptions of functionality units and effects in FEM.	2
(c) Completeness of flow	SeqD Step 3. The flow is ascertained from the descriptions of functionality units and class allocation principles.	2
(d) Minimality of participating objects	Crosschecks in FEM ensure this minimality.	2
(e) Minimality of operations	Implicit from (b)	2
(f) Minimality of flow	Implicit from (d)	2
	Total	12
	Strength of the NAVITA sequence diagram modelling technique	100

14.2.1.7 Evaluation of State Diagram

Table 14-7 gives a summary of the evaluation of NAVITA STD modelling technique.

Table 14-7 Evaluation of NAVITA STD Modelling Technique

Criterion	Modelling Guidelines	Rigour
(a) Completeness of states	States are identified through FEM and the state table. The analysis can only guarantee that essential states – such as after creation, before deletion – are present, but not much more.	1
(b) Completeness of events/transitions	Events and transitions are identified from FEM, in which FUs with an update effect are taken as events, and their effects are transitions from one state to another	2
(c) Minimality of states	Not provided.	0
(d) Minimality of events/transitions	FEM requires that at least a set of events are present so that the object/component can be created, updated and deleted.	1

	Total	4
	Strength of the NAVITA STD modelling technique	50

14.2.1.8 Evaluation of LFM

Table 14-8 gives a summary of the evaluation of the NAVITA LFM modelling technique.

Table 14-8 Evaluation of NAVITA LFM Modelling Technique

Criterion	Modelling Guidelines	Rigour
(a) Completeness of steps	LFD Steps 1 and 2. LFD steps are initially identified from the descriptions of functionality units in MFD. These steps are later cross-checked with the LSL and in particular USDM (USDM Step 2).	2
(b) Completeness of flow	Control structures of LFD and USDM are cross-checked.	2
(c) Minimality of steps	Implicit from (a).	1
(d) Minimality of flow	Implicit from (b).	1
	Total	6
	Strength of the NAVITA LFD modelling technique	75

14.2.2 Evaluation of Architecture

Chapter 12 and Chapter 13 provide detailed discussions on NAVITA Software Architecture. NAVITA provides a reference architecture (Figure 6.1) with great emphasis on user-driven component pluggability, reuse of high-grained business and user interface components, and seamless integration of distributed components. The use of Backbone component as a communication medium and repository for component services, and Application Manager to maintain components within the application are unique features of this architectural model.

14.2.2.1.1 Definition of Components

NAVITA does not provide a formal definition of the term "component." Instead it attempts to illustrate and discusses various key characteristics of NAVITA components. NAVITA distinguishes between two main types of components: boundary components and business components. All NAVITA components are service-oriented and technology-neutral. Boundary components are identified by

analysing the aspects of user-system interaction within the functionality unit, while business components are identified by looking at how related sets of functionality units affect cohesive sets of entity classes. The discussions on components focus on business components in NAVITA, as they are thought to be more important than boundary components. The unique feature of NAVITA is that components are identified at the point where the technical and business perspectives merge.

14.2.2.1.2 Identification and Verification of components

NAVITA provides concrete guidelines on how to identify business components and translate them into a design that can be implemented using common OO programming languages. NAVITA FEM plays a crucial role in identification of business components, whilst the protocol analysis provides operations that, together with IM fragments, LSL and USDM, form the interfaces of components.

14.2.3 Evaluation of System Development Process

Table 14-9 Evaluation of RSE System Development Process

The MAP Criteria	NAVITA
Feasibility Analysis	One of the first stages of development in NAVITA SDP, which emphasises the need to address CBSD-specific feasibility concern such as component availability in addition the concerns of traditional feasibility analysis.
Business Modelling	NAVITA emphasises the importance of BPM, while referring to BPMN for notation and SELECT Perspective for modelling technique.
Requirement Analysis	NAVITA provides extensive modelling technique for requirements analysis.
System Analysis	A number of modelling techniques are also provided for this analysis.
Logical Architecture	Specification of services provided by logical components is the main concern of logical architecture in NAVITA.
Physical Design	NAVITA provides detailed discussions on how to create components that address both business and technological constraints.
Component Search	This is a key part of NAVITA SDP, where references to existing material are provided.
Component Certification	This is part of Component Search and Acquisition.
Component Implementation	Assuming that applications will be developed using OO

	technologies, NAVITA provides design techniques to translate the logical design into component-based OO design. It does not however discuss implementation issues.
Application Assembly	Only references to existing material are provided.
System Testing	Only references to existing material are provided.
System Delivery	Only references to existing material are provided.

14.3 A Comparison of Evaluation Results

This section compares the evaluation results of NAVITA, presented in this chapter, with the evaluation results of the existing CBSD methods, presented in Chapter 4.

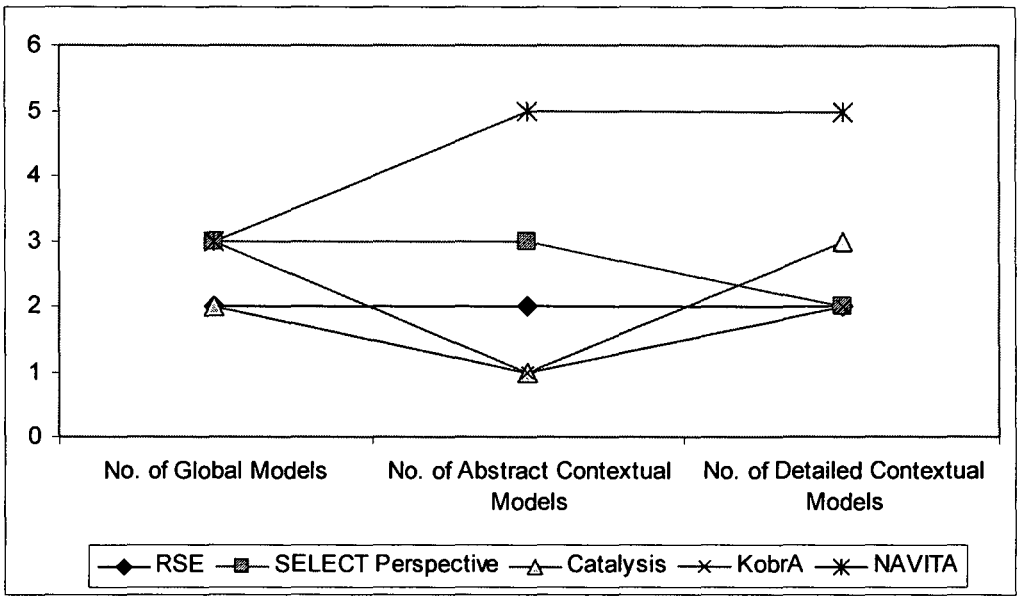
14.3.1 Correlations between the Three Elements of Methods

There are correlations between the three elements in both RSE and SELECT Perspective. Catalysis provides neither a detailed SDP nor a reference architectural model. As a result, only the correlations between modelling and general stages of SDP are clear. Although there is no reference architectural model in KobrA, correlations between the three elements are evident. It is fair to say that correlations between the three elements are most definitive in NAVITA.

14.3.2 Coverage Models

Chart B sums up the total number of important global and contextual models in existing CBSD methods and NAVITA. As far as global models are concerned, KobrA and NAVITA provide three global models each. Although SELECT Perspective provides four global models, Logical Data Model and Class Model occupy the same area of the IPI matrix, indicating overlaps between the two models. Both RSE and Catalysis lack a global model for interaction. However, none of the existing methods provide a sufficient number of abstract and detailed contextual models, meaning that there are few crosschecks between the global models of these methods. NAVITA makes substantial advances in this area of modelling by providing five abstract and five detailed contextual models. This is an important weakness of existing CBSD methods that NAVITA has tackled effectively.

Chart B Total Number of Models in Each Method



14.3.3 Relative Strengths of Modelling Techniques

Chart C compares the strengths of the global modelling techniques in existing CBSD methods and NAVITA. Techniques of global models are compared on the basis of the axis of the IPI Matrix that these models are identified with. As the chart indicates, NAVITA has made substantial progress in improving the rigour of global modelling techniques.

Chart C Relative Strengths of Comparable Modelling Techniques

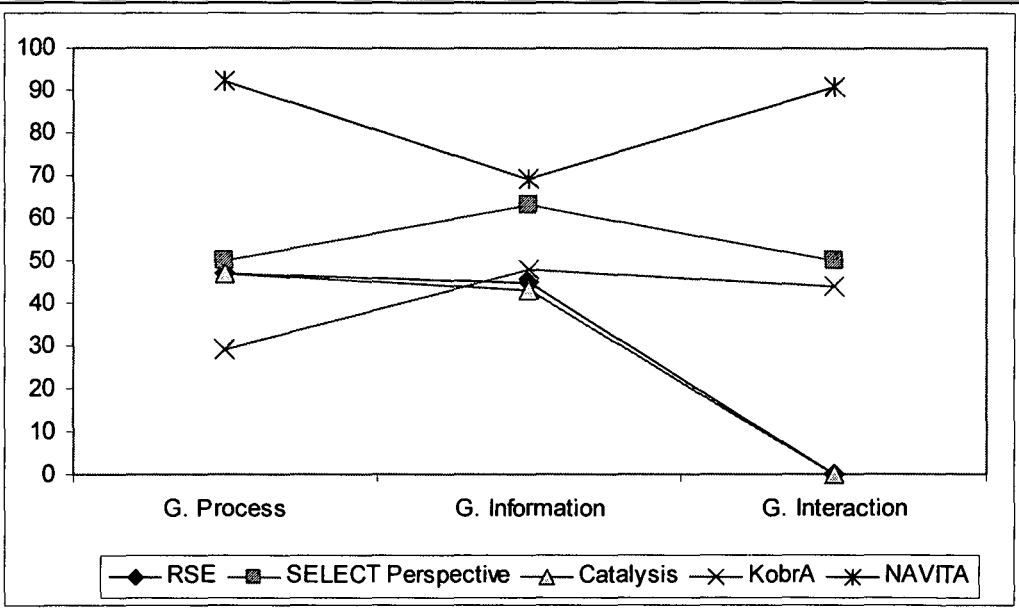
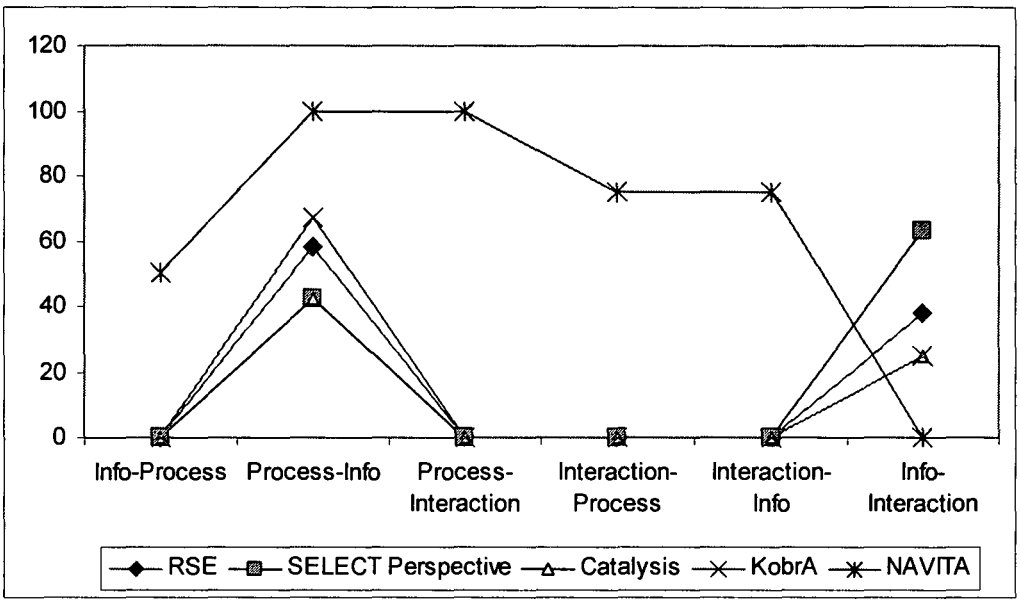


Chart D shows the relative strengths of the detailed contextual modelling techniques provided by existing CBSD methods and NAVITA. Modelling techniques of detailed context models are compared on the basis how these models are represented in the IPI matrixes; for example, models represented by arrows between the Information and Process axes pointing from the information model towards the process model. NAVITA, again, has made a leap forward in terms of improving the rigour of modelling techniques.

Chart D Relative Strengths of Comparable Contextual Modelling Techniques



14.3.4 Architectural Models

Catalysis and KobrA do not provide any reference architectural models, while RSE and SELECT Perspective offer architectural models that are largely reminiscent of OO systems: layered architecture. NAVITA provides an architectural model that reflects the nature of component-based applications, which is another improvement NAVITA has made over the existing methods.

14.3.5 SDP

SDP models of SRE and Perspective describe the CBSD process reasonably well, while Catalysis and KobrA put relatively little emphasis on this aspect of SDMs. NAVITA provides a SDP model that is in many ways similar to those in SRE and SELECT Perspective. This is an area of CBSD methods that is well-developed.

14.4 Summary

NAVITA has been evaluated by applying the MAP framework. Comparison of the evaluation results of NAVITA and existing CBSD methods shows that NAVITA is a complete CBSD method that overcomes many shortcomings of existing methods. NAVITA has more models, better coverage and stronger modelling techniques, while the architectural model closely reflects the nature of component-based applications. SDP of NAVITA has a similar coverage to SDPs of RSE and SELECT Perspective. These comparison results validate the second hypothesis of this research.

Chapter Fifteen

Research Methodology

15.1 Introduction

This research project is divided into two main stages. The first stage of the project is concerned with investigating existing CBSD methods, and with evaluating these methods using an evaluation framework. Based on the evaluation outcomes, stage two of the project focuses on the development of a holistic method to address shortcomings in the existing methods. Whilst the research is divided into two stages, this research is cumulative and evolutionary with a singular aim of contributing to the improvement of the general quality of CBSD methods. The research methodology used in this project can be summed up as follows:

Step 1. Initial investigation of existing CBSD methods

Step 2. Investigation of method evaluation approaches and their applicability to this research

Step 3. Development of a general theory of system development methods, and a comprehensive framework for evaluation of CBSD methods

Step 4. Evaluation of existing CBSD methods using the evaluation framework

Step 5. Validation of the evaluation results using a repeatability experiment

Step 6. Development of the new CBSD method

Step 7. Demonstration of the new CBSD method using a case-study

Step 8. Evaluation of the new CBSD method using the same evaluation framework used for the existing methods

15.2 Initial Investigation of CBSD methods

This research began with an initial investigation into the state of affairs of what was an increasingly prevailing approach to software development, namely, component-

based software development. The initial investigation had identified that, despite the major advances in software technologies concerning component-based software, systems development methods available at the time were largely undistinguishable from their predecessor OO methods such as OOSE (Jacobson, 1992) and UML. An informal assessment of the *status quo* revealed a number of weaknesses in these methods and a need for a new CBSD method. In addition to taking into account software component technologies, the new method would have to espouse the reuse philosophy in every aspect of software development, namely system modelling, software architecture and system development process, as called for by this new software development strategy. This initial investigation prompted this CBSD project.

As far as the research methodology is concerned, there are two main issues it has to address: first, objective evaluation of the existing CBSD methods; second, creation of a new CBSD method that demonstrably overcomes the shortcomings found in the existing methods.

The research methodology recognises that a comprehensive and objective evaluation of existing methods to expose strengths and weaknesses requires a consistent application of a rigorous set of criteria. Moreover, the subsequent demonstration of the relative merits of the new method must be based upon the evaluation results produced from application of the same set of criteria. Therefore, an appropriate use of a systematic evaluation approach has been imperative for the success of this research.

Since this research involves evaluation of existing CBSD methods, creation of a new method that improves upon the existing methods and demonstration of the relative merits of the new method over the existing ones, a plain evaluation approach is not sufficient. Any evaluation approach used in this research must also assist in the process of new method creation. It is important, therefore, to address the question of what constitutes a good method. In other words, a theory underpinning the evaluation and creation of SDM needs to be established before attempting to evaluate and create CBSD methods. Based on such a theory, a framework for evaluation and creation of CBSD methods may be derived and applied.

15.3 Investigation of Method Evaluation Approaches and their Applicability to this Research

In order to understand and evaluate the existing CBSD methods objectively, rigorously and consistently, a rational standard for evaluation is required, leading to an investigation of existing approaches to the evaluation of system development methods.

This research has identified two basic ways in which methods are evaluated by academics and practitioners alike. With the first approach, the evaluator uses a set of criteria that are often subjective, narrowly-focused, and somewhat random. The use of such an approach is justifiable if the evaluator's interest in the method is limited. For example, CASE-tool makers may only be interested in diagrammatic symbols, concepts and modelling rules; their interest in the rigour of modelling techniques, for instance, may be limited. Another approach to evaluation is the use of what are commonly known as evaluation frameworks, which range from a set of abstract principles about what a method should entail to more concrete criteria outlining the kinds of model that should be supported by a method. These evaluation frameworks tend to provide commentary on the frameworks' theoretical basis.

Since the object of evaluation in this research is not limited to how certain characteristics of a method rate, but rather to determining the quality of a method in its totality, evaluation of system development methods using a loosely arranged set of criteria would either fail or produce partial and inconclusive results. The strength of evaluation frameworks over list of criteria is in providing more systematic and organised approaches to understanding and evaluating methods. From the perspective of this research, which intends to investigate all major aspects of system development using CBSD approach, the evaluation approach using a framework is more appropriate and more likely to produce useful results.

This research has identified two evaluation frameworks that are relevant to this project, namely, NIMSAD and Wieringa's framework. As discussed in Chapter 2, NIMSAD does provide a good theoretical basis for understanding and evaluating system development methods, but suffers from being too generic and process-oriented. It nevertheless provides the foundation for the new evaluation approach (Section 3.2). Wieringa's framework provides a similar kind of evaluation criteria,

but is focused exclusively on requirement specification techniques. The new framework takes onboard elements of Wieringa's framework.

15.4 Development of an SDM theory and the MAP Framework

The first major step of this research is to establish a theoretical framework to explain what a method should include and how it should be evaluated. By drawing lessons from systems thinking, NIMSAD, Wieringa's framework and other criteria suggested by various authors, this research establishes a theory of system development method and, based on the theory, a novel approach to understanding and evaluating existing and new system development methods, named the MAP framework. This framework, therefore, serves as the archetypal criteria for the rest of the research. This framework has been established before properly evaluating existing methods and creating a new method. In this way subsequent evaluation of existing methods and the new method can be both objective and consistent.

15.5 Evaluation of existing methods using the new framework

Once the framework for evaluation has been established, existing CBSD methods are studied again in greater detail. This involves summarising important features of the methods. A survey paper (Appendix II) has been prepared based on this study[9]. The methods are then evaluated using the MAP evaluation framework.

15.6 Validating the evaluation results using an experiment

One of the questions that arose from an attempt to publish the evaluation framework was whether the evaluation results would be repeatable, i.e. if other evaluators were to use the framework to evaluate the same method, would they arrive at the

[9] Perhaps an informal part of the research methodology is the author's involvement with some teaching activities both within the London Metropolitan University and outside of it. Part of this paper, for instance, is an edited version of some lecture notes used for teaching component-based development and SELECT Perspective for final year BSc degree students. Interactions with students during lectures and tutorials on a wide range of subjects from UML to SSADM were a source of insight and inspiration for this author.

conclusion this research had reached. In order to establish the repeatability of the evaluation framework, an experiment was set up involving final year degree students taking the Advanced Systems Analysis and Design unit at London Metropolitan University. All students were exposed to key SSADM and UML techniques and were asked to carry out the evaluation. The majority of the participants came to the same conclusion that SSADM models have a better coverage and inter-model checks than UML, confirming that the framework, to a large extent, is repeatable.

15.7 Development of the new CBSD method

Equipped with a general outline of what a good method should entail and the evaluation results of existing methods, the project has then proceeded to create a new method that takes into account both strengths and weakness of existing methods. The process used is iterative, not recursive, since the evaluation framework provides guidelines for how to move from one stage to another. The main advantage of using a comprehensive evaluation framework like the one suggested by this research is this: the framework not only helps identify positive features of various methods, but it also helps determine the way that weaknesses should be tackled. Furthermore, this framework, with its great emphasis on integration of different aspects of a method, helps in the process of assembling different parts of the method into a single cohesive one. The result is NAVITA, a systematic synthesis of good features of existing CBSD and non-CBSD methods, with novel insights into the nature of system development using the component-based approach.

15.8 Demonstration of the new method using a common case-study

A case study is used throughout the research for two main purposes: firstly, to discuss and present modelling concepts of NAVITA, and secondly, to illustrate how the modelling approach suggested by the new method will work as a whole. The second point is also an indication of the completeness of the new method in the sense that it can be used from requirements analysis down to implementation-specific system design. The specific application chosen for this study is a simple student library, named LibrInfoSys. A student library case study is chosen because it is a popular one used by many authors on system development methods such as

(Atkinson, 2002), and one with small, but reasonably complex, business constraints found typically in many business information systems.

15.9 Evaluation of the new CBSD method

After NAVITA has been constructed and demonstrated using the case study, it is formally evaluated against the evaluation framework used to assess the existing methods. The research, therefore, deploys the same yardstick to measure all methods; any claim made for or against the new method is grounded in clear evidence, objectivity and consistency.

15.10 Methodological Issues

The methodology of this research has the following limitations:

- The software architecture suggested by NAVITA should have been demonstrated with the use of either a prototype including the major components or an entire application. However, due to limitation of time and scope of the research, a decision was made against the plan to develop such a prototype.
- The formula used to calculate strength of a modelling technique can be improved; see Section 16.3.
- In the project proposal (Appendix VI), it was suggested that the new method would be demonstrated by applying it to applications in three different domains. This became unfeasible because of time constraints and inappropriate because NAVITA was created for use in development of enterprise ISs; see Section 5.3. The expert review approach was not used due to its subjectivity; see 2.3.1.

Chapter Sixteen

Conclusions, Contributions and Areas for Further Research

16.1 Research Conclusions

The first part of this research is mainly concerned with surveying and evaluating CBSD methods. A theoretical and comprehensive evaluation of these methods requires a rigorous application of detailed evaluation criteria which most existing approaches to evaluation of SDM do not provide. This research proposes a novel evaluation approach to systematic evaluation of SDM, and Component-based SDM in particular, which is then applied to the CBSD methods surveyed. The evaluation has identified strengths and weaknesses of these methods, and based on the findings, a new method is proposed. The new method is then evaluated by applying the same criteria and rigour used for existing methods. A comparison of evaluation results shows that the new method overcomes major weaknesses in previous methods, in particular in the area of system modelling. Therefore, the two objectives of this research, outlined in Section 1.3 have been achieved.

16.2 Research Contributions

In fulfilling its objectives, this research has made four key contributions for the advancement of CBSD methods:

(i) Detailed Analytical Survey of CBSD Methods

A detailed investigation of four publicly available CBSD methods has been carried out in this research and summaries of their key features are presented in journal paper format in Appendix II. The survey mirrors the development

and the *status quo* of CBSD methods, which can be used as a starting point for further exploration of CBSD methods by researchers and practitioners alike.

(ii) The MAP framework and its application

Using systems thinking theory and some well-established generic evaluation frameworks as the starting point, this research has provided a theory of SDMs and a novel approach to evaluating system development methods. The MAP framework is superior to existing evaluation approaches in a number of aspects. It covers all major technical issues of system development methods, with a strong emphasis on evaluation of models and modelling techniques. IPI Matrix, underpinned by a general theory of system development methods, provides a conceptual framework for classifying and analysing relationships and dependencies between models suggested by system development methods of all kinds. Using this unique matrix, models can be analysed and evaluated diagrammatically by looking at how they help analysts gain a complete and consistent understanding and projection of a problem situation. Most existing evaluation criteria deal largely with global models, but the MAP framework places a unique emphasis on the importance of contextual models in ensuring that the global models are internally and externally consistent. In other words, models do not contradict each other and fulfil users' real requirements. Formulation of criteria for evaluation of individual models is based on key quality characteristics of requirements specifications. These criteria are uniform and consistent across all models. The framework highlights the importance of evaluating software architecture, and indicates mechanisms for the evaluation. Criteria for evaluation of the system development process are a customisation of an established framework. Therefore, evaluation criteria generated by the MAP framework are systematic, detailed and highly organised. Rigorous application of these criteria leads to a clear indication of the overall quality of a method.

The MAP framework is then successfully applied to existing CBSD methods. The evaluation uncovers the following key weaknesses:

- Existing CBSD methods tend to have few abstract and detailed contextual models. As a result, global models have few crosschecks between them.
- Existing CBSD methods do not place enough emphasis upon the need for capturing user-system interaction at the global level, and integrating it with other models.
- Rigour of modelling techniques is rather weak because of the lack of crosschecks between global models and because users are not involved in modelling. The MAP framework shows that the use of more formal specification techniques does not automatically lead to production of specifications that are internally and externally consistent.

The evaluation results confirm the first hypothesis of this research.

Outside the mandate of this research, the MAP framework has been applied to a popular Structured method, SSADM, and an OO method, UML (Bielkowicz and Tun, 2003; Bielkowicz et al, 2002). Evaluation of these methods using the MAP framework confirms the suspicion of some researchers that UML models are fragmented, with little crosschecks between global models. In this respect, SSADM is a highly sophisticated method.

This evaluation shows that the MAP evaluation framework can also be used by others to evaluate various kinds of SDMs. An experiment involving independent practitioners of the framework supports the repeatability of the framework.

(iii)New CBSD Method: NAVITA

Application of the framework to CBSD and non-CBSD methods leads to a critical understanding of what makes a good SDM. This gives the impetus to synthesise various elements of existing methods with novel insights into system modelling in order to create this new CBSD method.

SDP of NAVITA ensures that models are produced only when necessary; the development of models is not required when components are available. NAVITA recognises that specifications of components must be produced before they are designed for implementation. NAVITA models allow analysts

to produce requirements specifications, component specifications and component designs only when needed. Reference Architecture for software applications suggested by NAVITA enables a vast array of components to be slotted into the application through its Application Manager component. Components within an application can communicate with each other, in a technology-independent and location-transparent manner, through the Backbone component. Practitioners of the component-based approach can benefit from the rigorous modelling, flexible architectural model, and SDP of NAVITA.

Models and modelling techniques suggested by NAVITA are based on lessons learnt from evaluation of existing CBSD and non-CBSD methods alike. This research has synthesised elements of existing methods with unique insights into creating models and modelling techniques for the new method. Context Diagram, as it is traditionally used, is examined critically and important improvements are suggested in the semantically-richer NAVITA Context Diagram. NAVITA has tackled the confusion with loose granularity of use cases by providing the concept of "functionality units." Hierarchical nature of processes is captured using a separate model, named LFD. NAVITA puts a unique emphasis on user-system interaction by providing a static and a dynamic model, LSL and USDM, to analyse this important and often ignored aspect of the system. Information modelling is supported by FEM, which serves as a crucial crosscheck between the MFD and IM. NAVITA IM is further crosschecked with LSL through Relational Data Analysis, enabling the users to validate the models being developed. Protocol Model is another exclusive feature of NAVITA used for the analysis of component communication. The component modelling technique provided by NAVITA is methodical in ensuring that the business components satisfy both technical and business constraints. FEM plays the key role in identifying components in this modelling technique. Discussion of component modelling covers how the components identified can be implemented using standard OO technologies. NAVITA models are tightly knit and modelling techniques are detailed.

NAVITA is then subjected to the same kind of evaluation as the existing CBSD methods. The evaluation results show that NAVITA is demonstrably

better than existing methods, especially in the area of system modelling. This is the confirmation for the second hypothesis of this research.

(iv)Principles for Rational Allocation of Class Operations

Any rigorous class modelling must deal with two questions. First, for a given functionality unit, how can analysts determine whether an operation should be allocated to a class? Second, how can analysts determine the nature of such an operation? This research has investigated various class modelling techniques suggested by popular OO methods, and proposes a rational and deterministic approach to allocation of class operations in terms of two principles (Appendix III). These principles may be beneficial to practitioners of both OO and CBSD approaches.

16.3 Areas for Further Research

This research raises issues for further investigation that lie beyond the scope of this project. These include:

(i) Further Refinements of The MAP Framework

In this research, the strength of a modelling technique is derived by calculating the extent to which the total rigour fulfils the maximum possible rigour for the technique; see 3.4.3.1. However, in each model some model elements are more important than others. For example, Content elements of a global model may be more important than Structural elements; see 3.4.1.4. This means that in a global information model, entities may be regarded as more important than entity relationships. Giving an equal weight to all elements of all models penalises, to a varying extent, those global models that score well for criteria concerning those important model elements, but not for the many less important model elements. Since the number of elements in comparable models does not vary much and most elements have a similar level of importance, specific weightings have not been used in this research. For future application of the MAP framework, this can be improved by giving, for example, criteria for Content/Functional elements a weighting of 5, Structural 3 and Overlap 1.

Furthermore, strengths of all modelling techniques may be added together to give a single score for all modelling techniques in an SDM. Here again, different weightings may be used: global models may be regarded as more important than detailed contextual models, which in turn may be considered more important than abstract contextual models. A formula such as the one shown below may be used.

SAMT = (GMW × (Σ GMT Strength)) + (DCW × (Σ DCMT Strength)) + (ACW × (Σ ACM))

Where, SAMT = Strength of All Modelling Techniques,

GMW = Global Models Weighting = 5,

GMT Strength = Strength of a Global Modelling Technique,

DCW = Detailed Contextual Models Weighting = 3,

DCMT Strength = Strength of a Detailed Contextual Modelling Technique,

ACW = Abstract Contextual Models Weighting = 1,

ACM = Abstract Contextual Model.

(ii) Tools for the MAP framework and NAVITA

Production of IPI Matrix, generation of criteria for global and contextual models, and analyses of the matrix and the rigour of modelling techniques can be supported by an automated tool. NAVITA CASE tool may also be created by customising an open source UML tool such as ArgoUML (Tigris, n.d.).

(iii)Higher-level Functionality Modelling

There are two main ways in which behaviour of a system or part of a system is usually described: using procedure-oriented diagrams such as UML Activity Diagram and using state-oriented diagrams such as UML Statechart Diagram.

Many authors of SDMs claim that dynamic global models can be produced using diagrams such as UML Statechart (OMG, 2003), Operation Spec/State chart (Coleman et al, 1994) and BPM (Allen and Frost, 1998). These authors often give simple examples to justify their claims. Fusion, for example,

demonstrates the Operation State/Spec using a simple ATM system. The only problem is that the diagrams these authors tend to show are either too simplistic and/or do not capture the business constraints well.

In the course of development of NAVITA, it was hoped that it would be possible to show the logical order of FUs, using the same concepts deployed to show the logical order of steps within a functionality unit, i.e. sequence, selection and iteration used in LFM. When a 'Higher-level Functionality Model' as it would be called, was produced for the entire library system, it was discovered that these concepts cannot capture accurately the constraints applicable to the high level ordering of functionality units. State-oriented diagrams do not capture the constraints well either.

The diagnosis of the problem by this author is that, when dealing with state-dependent behaviour, there tends to be an obvious symmetry if the analysis is focused upon a single significant part of the system. Therefore, it is not problematic to show state-dependent behaviour of a system from the perspective of an individual entity (ELH) or class (state diagram). The general pattern is that each entity or object is first created, before going through a midlife cycle of state changes, after which the entity or object is destroyed. However, if the analysis is to show the behaviour of the system from the perspective of more than one object or entity, the simple regularity disappears, rending most control constructs – sequence, selection, control, parallelism, sub-state etc – incapable of expressing the complex dependencies. Interestingly, this confirms the suggestion by the MAP framework that dynamic models are suitable only for contextual modelling, not global modelling; see 3.4.1.6.

Perhaps this issue can be tackled using a formal or mathematical specification language. Questions of whether it may be possible to express in a single diagram dependencies between functionality units of an entire system, and if so how it can be done, need further investigation.

(iv)Prototype for the NAVITA Software Architecture

A prototype of the Reference Architecture for software suggested by NAVITA should be implemented. This will demonstrate how three key elements of modern software architecture – software plug-in technology, middleware architecture and HTTP-like session-based software communication – can be integrated to delivery flexible architecture for reusable components.

References

17.1 References

Allen, P. and Frost. S., 1998. *Component-Based Development for Enterprise Systems: Applying the SELECT Perspective*. Cambridge: Cambridge University Press/SIGS Books.

Allen, R. and Garlan, D., 1994. Formalising Architectural Connection. In: *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, 71-80.

Ambler, S., 1998. *Process Patterns: Delivering Large-scale Systems Using Object Technology*. Cambridge: Cambridge University Press.

Ambler, S., 1999. *More Process Patterns: Delivering Large-scale Systems Using Object Technology*. Cambridge: Cambridge University Press.

Aoyama, M., 1997. Process and Economic Model of Component-Based Software Development. In: *Proceedings of the 5th International Symposium on Assessment of Software Tools (SAST), 3-5June 1997, Pittsburgh, PA*. IEEE Computer Society Press, 100-103.

Aoyama, M., 1998a. Component-Based Software Engineering: Can it Change the Way of Software Development? *In: Proceedings of 20th International Conference on Software Engineering Vol. II, 19–25 April 1998, Kyoto*. IEEE Computer Society Press, 24-27

Aoyama, M., 1998b. New Age of Software Development: How Component-Based Software Engineering Changes the Way of Software Development. *In: Proceedings of the 1st workshop on Component Based Software Engineering*, April, 1998 Kyoto. Available from: http://www.sei.cmu.edu/pacc/icse98/papers/p14.html [Accessed 09 May 2005]

Armour, F. and Miller, G., 2001. *Advanced Use Case Modeling: Software Systems Vol. 1*, London: Addison-Wesley.

Ashby, W. R., 1947. Principles of the Self-Organizing Dynamic System, *Journal of General Psychology*, 37, 125-128.

Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wust, J. and Zettel, J., 2002. *Component-Based Product Line Engineering with UML*. Harlow: Pearson Education Ltd.

Avison, D. E., and Fitzgerald, G., 2003. *Information Systems Development: Methodologies, Techniques and Tools 3rd ed.* Berkshire, England: McGraw-Hill.

Avison, D. E., and Fitzgerald, G., 1995. *Information Systems Development: Methodologies, Techniques and Tools 2nd ed.* Berkshire, England: McGraw-Hill.

Bachmann, F., Bass, L., Chastek, G., Donohoe, P. and Peruzzi, F., 2000. *The Architecture Based Design Method*, Technical Report, Software Engineering Institute, Carnegie Mellon University. Available from: www.sei.cmu.edu/pub/documents/ 00.reports/pdf/00tr001.pdf [Accessed 13 May 2005]

Barbacci, M. R., Klein, M. H. and Weinstock C. B., 1997. *Principles for Evaluating the Quality Attributes of a Software Architecture*. Technical Report, Software Engineering Institute, Carnegie Mellon University. Available from: http://www.sei.cmu.edu/publications/documents/03.reports/03tn012.html [Accessed 13 May 2005]

Bashir, A., 2003. A Paradigm Shift: Moving Onto Components; A Management Perspective. *Journal of Conceptual Modelling* [online], 28. Available from: http://www.inconcept.com/JCM/May2003/Bashir.html [Accessed 12 April 2005].

Basili, V. S., Shull, F. and Lanubile, F., 1999. Using Experiments to Build a Body of Knowledge. *Ershov Memorial Conference, Akademgorodok, Novosibirsk, Russia, July 6-9*. 265-282.

Basili, V. R., 1993. The Experimental Paradigm in Software Engineering. In: *Experimental Software Engineering Issues: Critical Assessment and Future Directives, Proceedings of Dagstuhl-Workshop, September 1992*. Available from: http://www.cs.umd.edu/~basili/papers.html [Accessed 09 May 2005]

Bass, L., Clements, P. and Kazman, R., 1998. *Software Architecture in Practice*. Addison-Wesley Longman.

Beck, K., 1999. *Extreme Programming Explained: Embracing Change*. Harlow: Addison Wesley.

Bellin, D. and Simone, S. S., 1997. *The CRC Card Book*. Reading, Mass: Addison-Wesley.

Bennett, S., McRobb, S. and Farmer, R., 2001. *Object-oriented Systems Analysis and Design Using UML 2nd edition*. McGraw-Hill Education.

Bentley, C., 1997. *Introducing SSADM 4+*. London: Stationery Office.

Bertalanffy, L. von., 1968. *General System Theory: Foundations, Development, Applications*. London: Allen Lane.

Bertolino, A. and Polini, A. A Framework for Component Deployment Testing. In: *Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon, May 03 - 10, 2003*. 221 – 231.

Bielkowicz, P. and Tun, T. T., 2003. A Critical Assessment of UML Using An Evaluation Framework. In: *International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design (EMMSAD'03) Velden, Austria June 16-17*

Bielkowicz, P., Patel, P. and Tun, T. T., 2002. Evaluating Information Systems Development Methods: A New Framework. In: *Proceeding of the 8th International Conference on Object-Oriented. Information Systems, Montpellier, France, September 2-5*, 311-322.

Bielkowicz, P, and Tun, T. T., 2001. A Comparison and Evaluation of Data Requirement Specification Techniques in SSADM and the Unified Process. In: *Proceedings of Advanced Information Systems Engineering, 13th International Conference, CAiSE 2001, Interlaken, Switzerland, June 4-8, 2001*, 46-59.

Bittner, K. and Spence, I., 2002. *Use Case Modelling*. Addison Wesley.

Bjorn-Andersen, N., 1984. Challenge to Certainty. In: Bemelmans T. M. A., ed. *Beyond Productivity: Information Systems Development for Organisational Effectiveness*. Amsterdam: North Holland.

Boillot, M. H., Gleason, G. M. and Horn, L.W., 1995. *Essentials of Flowcharting*. William C. Brown.

Booch, G., 1991. *Object Oriented Design: With Applications*. California; Wokingham: Benjamin/Cummings.

Booch, G., 1994. *Object-Oriented Analysis and Design with Applications 2nd edition*. California; Wokingham: Benjamin/Cummings.

Booch, G., Rumbaugh, J. and Jacobson, I., 1999. *The Unified Modeling Language User Guide*, Harlow: Addison-Wesley.

Boyd, R., Gasper, P. and Trout, J.D., 1991. *The Philosophy of science*. Massachusetts Institute of Technology Press.

Brinkmeyer, H. A New Approach to Component Testing. In: *Proceedings of the Conference on Design, Automation and Test in Europe – vol 1. March 07 – 11*. 534 – 535.

Brown, A. W., 2000. *Large Scale Component Based Development*. Prentice Hall.

Brown, A., 1996. *Foundations of Component-Based Software Engineering*. IEEE Computer Society Press.

Brown, A., ed, 1996. *Component-based Software Engineering: Selected Papers from the Software Engineering Institute*, IEEE Computer Society Press.

Brown, S., Fauvel, J. and Finnegan, R., ed (1989 reprint). *Conceptions of Inquiry: A Reader*. Routledge/The Open University Press.

Bubenko Jr., J. A., 1986. *Information System Methodologies – A Research View*: in Olle et al (1991).

Burns, A. and Wellings, A., 2001. *Real-time Systems and their Programming Languages* 3rd ed. Essex, England: Pearson Education Limited.

Business Process Management Initiative (BPMI), 2004. *Business Process Modeling Notation (BPMN) Version 1.0*. Available from: http://www.bpmn.org [Accessed 12 April 2005]

Cameron J. R., 1989. *JSP & JSD: The Jackson Approach to Software Development 2nd edition*. Washington DC: IEEE CS Press.

Campbell, G. H. Jr, 1999. Adaptable Components. In: *Proceedings of the 21st International Conference on Software Engineering, Los Angeles, California, USA, May 16 - 22, 1999*, 685 – 686.

Casson, C., 2000. *Leveraging the Benefits of Business Process Modelling for IT Development to Improve the Alignment Between Business Requirements and Supporting IT Systems*. MSc Thesis, London Guildhall University.

Catchpole, C. P., 1987. Information Systems Design for the Community Health Services. PhD Thesis, Aston University, Birmingham.

Chávez, A., Tornabene, C. and Wiederhold, G., 1998. Software Component Licensing: A Primer. *IEEE Software*. 15(5), 47-53.

Cernosek, G. and Naiburg, E., 2004. *Value of Modelling*. IBM.

Checkland, P., 1999. *Systems Thinking, Systems Practice: Includes a 30-year Retrospective*. West Sussex, England: Wiley.

Cheesman, J. and Daniles, J., 2001. *UML Components: A Simple Process for Specifying Component-Based System*. Boston, MA: Addison-Wesley.

Lüer, C. and Rosenblum, D.S., 2001. WREN --- An Environment for Component-Based Development. In: *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001, Vienna, Austria*, 207-217

CIDE, 1995. *Cambridge International Dictionary of English*. Cambridge: Cambridge University Press.

Coad, P. and Nicola, J., 1993. *Object-Oriented Programming*. London: Prentice-Hall.

Cockburn, A., 2000. *Writing Effective Use Cases*. Addison-Wesley.

Coleman, D., Arnold, P., Bodoff, S., Dollin, D., Gilchrist, H., Hayes, F. and Jeremas, P., 1994. *Object-Oriented Development: The Fusion Method*. Prentice-Hall International.

Collins-Cope, M. and Matthews, H., 2001. A Reference Architecture for Component Based Development. In: *Proceedings of the 6th International Conference on Object Oriented Information Systems, London, UK*, pp. 225-237.

ComponentSource, n.d. www.componentsource.com [Accessed 15 December 2004]

Coplien J. O. (1997). Idioms and Patterns as Architectural Literature. *IEEE Software*, 14(1), 36-42.

Crnkovic, I. and Larsson, M., 2002. Challenges of Component-Based Development, *Journal of System and Software*, 61, 201-212.

Date, C. J., 1995. *An Introduction to Database Systems*, 6th ed. Addison-Wesley.

Davis, M. J. Adaptable, Reusable Code. In: *Proceedings of the 1995 Symposium on Software Reusability, Seattle, Washington, United States, April 29 – 30, 1995*. 38 – 46.

Derr, K., 1995. *Applying OMT: A Practical Step-by-step Guide to Using the Object Modeling Technique*, New York: SIGS Books.

DSDM Consortium, 2000. DSDM *and Component-Based Development*. DSDM Whitepaper.

D'Souza, D. F. and Wills, A. C., 1999. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison Wesley.

Edelstein, H., 1994. Unraveling Client/Server Architecture. *DBMS*, 34(7).

Eriksson, H-E and Penker, M., 1998. *UML toolkit*. Chichester: Wiley.

Evaristo J. R. and Karahanna E., 1997. Is North American IS Research Different from European IS Research? *The DATA BASE for Advances in Information System*, 28(3), 32-43.

Firesmith, D., Henderson-Sellers, B., Graham, I., 1997. *The OML Reference Manual*. New York: SIGS Books.

Fowler, M. and Scott, K., 1997. *UML Distilled: Applying the Standard Object Modeling Language*. Harlow: Addison-Wesley.

Gallagher, B. P., 2000. *Using the Architecture Tradeoff Analysis Method℠ to Evaluate a Reference Architecture: A Case Study*. Carnegie Mellon University, Software Engineering Institute. Available from: http://www.sei.cmu.edu/publications/pubweb.html/ [Accessed 12 April 2005]

Galliers, R. D. and Land, F. F., 1987. Choosing Appropriate Information Systems Research Methodologies. *Communications of the ACM*, 30(11), 900-902.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J., 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, Mass: Addison-Wesley.

Garlan, D. and Perry, D. E., 1995. Introduction to the Special Issue on Software Engineering (Guest Editorial). *IEEE Transactions on Software Engineering*, 21(4), 269-274.

Glinz, M., 2000. Problems and Deficiencies of UML as a Requirements Specification Language. In: *Proceedings of the Tenth International Workshop on Software Specification and Design. San Diego, USA, November 5-7, 2000*. 11-22.

Available from: http://www.ifi.unizh.ch/groups/req/staff/glinz/activities.html [Accessed 09 May 2005]

Glinz, M., Berner, S., Joos, S., Ryser, J., Schett, N. and Xia, Y., 2001. The ADORA Approach to Object-Oriented Modeling of Software. In: *Proceedings of the Advanced Information Systems Engineering, Proceedings of CAiSE 2001, Interlaken, Switzerland, June 4-8, 2001.* 76-92. Available from: http://www.ifi.unizh.ch/groups/req/staff/glinz/activities.html [Accessed 09 May 2005]

Gomaa, H., 2000. *Designing Concurrent, Distributed and Real-time Applications with UML.* Addison-Wesley.

Goodland, M. and Slater, C., 1995. *SSADM version 4: A Practical Approach,* London: McGraw-Hill.

Graham, I., Henderson-Sellers, B. and Younessi, H., 1997. *The OPEN Process Specification* Addison-Wesley.

Hares, J. S., 1994. *SSADM Version 4: The Advanced Practitioner's Guide.* Chichester, UK: John Wiley & Sons.

Heineman, G. and Councill, W., 2001. *Component-Based Software Engineering – Putting the Pieces Together.* Addison-Wesley.

Heineman, G. T, 1998. Adaptation and Software Architecture. In: *Proceedings of the 3rd International Workshop on Software architecture, Orlando, Florida, USA November 01 - 05, 1998.* 61 – 64.

Henderson-Sellers, B., Simons, T. and Younessi, H., 1998. *The OPEN Toolbox of Techniques* Addison-Wesley.

Hong, S., Goor, G. van den and Brinkkemper, S. A Formal Approach to the Comparison of Object-Oriented Analysis and Design Methodologies. *In: Proceedings of the 26th Hawaii International Conference on System Sciences, January 1993.* 689-698.

Hutt, A. T., ed, 1994. *Object Analysis and Design: Comparison of methods.* OMG Wiley/QED Publication.

Ince, D., 2003. *Developing Distributed and E-commerce Applications.* Addison-Wesley.

International Standardization Organisation, ISO9241, 1998. *Ergonomic Requirements for Office Work with Visual Display Terminals (VDTs), Part 11, Guidance on Usability,*

Jacobson, I., Booch, G. and Rumbaugh, J., 1999. *The Unified Software Development Process.* Addison-Wesley.

Jacobson, I., Christerson, M., Jonsson, P. and Overgaard, G., 1997. *Software Reuse: Architecture, Process and Organization for Business Success.* ACM Press/Addison-Wesley.

Jacobson, I., Ericsson, M. and Jacobson, A., 1994. *The Object Advantage: Business Process Reengineering with Object Technology*. ACM Press/Addison-Wesley.

Jacobson, I., Christerson, M., Jonsson, P. and Overgaard, G., 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach (revised)*. ACM Press/Addison-Wesley.

Jakes, J. M. and Yoche, E. R., 1989. Basic Principles of Patent Protection for Computer Software. *Communications of the ACM*. 32(8), 922-924.

Jayaratna, N., 1994. *Understanding and Evaluating Methodologies (NIMSAD): A Systemic Framework*, McGraw-Hill.

Kazman R., Abowd G., Bass L., and Clemens P. (1996). Scenario-Based Analysis of Software Architecture. *IEEE Software,* November 1996.

Kircher, M. and Jain, P., 2004. *Pattern-Oriented Software Architecture: Patterns for Distributed Services and Components*. John Wiley and Sons.

Kitchenham, B. and Pickard, L.,1995. Case Studies for Methods and Tool Evaluation. IEEE Software. 12(4), 52-62.

Kitchenham, B., 1992. A Methodology for Evaluating Software Engineering Methods and Tools. *Experimental Software Engineering Issues*. 121-124.

Kitchenham, B., Pfleeger, S. L. and Fenton N. E., 1995. Towards a Framework for Software Measurement Validation. *IEEE Transactions on Software Engineering*. 21(12), 929-943.

Kitchenham, B., Pfleeger, S. L., Pickard, L., Jones, P., Hoaglin, D. C., El Emam, K. and Rosenberg, J. 2002. Preliminary Guidelines for Empirical Research in Software Engineering. *IEEE Transactions on Software Engineering*. 28(8), 721-734.

Kotonya, G. and Sommerville, I., 1998. *Requirements Engineering: Process and Techniques*. West Sussex, England: John Wiley & Sons.

Kozaczynski, W. and Booch, G., 1998. Component-Based Software Engineering (Guest Editors' Introduction). *IEEE Software*. 15(5). 34-36.

Land, F., 1998. A Contingency Based Approach to Requirements Elicitation and Systems Development, *Journal of Systems Software*. 40(1), 183-184.

Leach, R., 1997. *Software Reuse: Methods, Models, and Costs*. London: McGraw-Hill.

Leffingwell, D. and Widrig, D., 2003. *Managing Software Requirements: A Use Case Approach*. Addison Wesley.

Loucopoulos, P. and Karakostas, V., 1995. *System Requirements Engineering*. McGraw-Hill.

Lowe, J., 2002. *A Survey of Metaphysics*. Oxford: Oxford University Press.

Microsoft Digital Network (MSDN) Microsoft Visual Basic Developer Center, Available from: http://msdn.microsoft.com/vbasic/default.aspx [Accessed 12 April 2005]

Monroe, R. T., Kompanek, A., Melton, R. and Garlan, D., 1997. Architectural Styles, Design Patterns, and Objects. *IEEE Software*, 14(1), 43-52.

Mozilla FireFox Available from: http://www.mozilla.org/products/firefox/ [Accessed 12 April 2005]

Olle, T.W., Hagelstein, J., Macdonald, I. G., Rolland, C., Sol, H. G., Van Assche, F. J. M. and Verrijn-Stuart, A. A., 1991. *Information Systems Methodologies: A Framework for Understanding* 2nd ed, International Federation for Information Processing/Addison-Wesley.

OMG, 2003. OMG UML Specification version 2. Available from: www.omg.org/uml [15 May 2005]

OMG, 1999. OMG UML Specification version 1.3. Available from: www.omg.org/uml [15 May 2005]

Orfali, R. and Harkey, D., 1997. *Client/Server Programming with Java and Corba 2nd ed.* John Wiley & Sons.

Pfleeger, S. L., 1995. Experimental Design and Analysis in Software Engineering. *Annals of Software Engineering vol. 1*. 219-253.

Pfleeger, S. L., 1999. Albert Einstein and Empirical Software Engineering. *IEEE Computer*. 32(10), 32-37.

Pooley, R. and Stevens, P., 1999. *Using UML: Software Engineering with Objects and Components*. Harlow: Addison-Wesley.

Popper, C., 1972. *The Logic of Scientific Discovery* 6th impression revised. London : Hutchinson.

Poseidon, Available from: http://www.gentleware.com/ [Accessed 14 April 2005]

Pressman, R. S., 2005. *Software Engineering : A Practitioner's Approach*, 6th ed. London: McGraw-Hill.

Bahsoon, R. and Emmerich, W., 2003. Evaluating Software Architectures: Development, Stability and Evolution. In: *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications*. (Also: UCL-CS Research Notes RN/02/25) Available from: http://www.cs.ucl.ac.uk/staff/r.bahsoon/Publications.htm [Accessed 13 May 2005]

Ramachandran, M., 2003. Testing Reusable Software Components from Object Specification. *ACM SIGSOFT Software Engineering Notes*. 28(2). 18.

Rational Rose, Available from: http://www.gentleware.com/ [Accessed 14 April 2005]

Rosenberg, D. and Scott, K., 1999. *Use Case Driven Object Modeling with UML: A Practical Approach.* Addison Wesley.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W., 1991. *Object-Oriented Modeling and Design*, Englewood Cliffs, New Jersey: Prentice-Hall.

Rumbaugh, J., Jacobson, I. and Booch, G., 1999. *The Unified Modeling Language Reference Manual*, Harlow: Addison-Wesley.

Sametinger, J., 1997. *Software Engineering with Reusable Components.* London: Springer.

Sanskrit Dictionary, n. d. http://www.alkhemy.com/sanskrit/dict/ [Accessed 14 April 2005]

SCIPIO, n. d. SCIPIO Method. Available from http://www.users.globalnet.co.uk/~rxv/scipio/ [Accessed 16 May 2005]

Shaw, M. and Garlan, D., 1996. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall.

Shaw, M., 1995. Comparing Architectural Styles. *IEEE Software*. 12(6), 27-41.

Shneiderman, B. and Plaisant, C., 2004. *Designing the User Interface: Strategies for Effective Human-Computer Interaction 4th ed.* Pearson/Addison Wesley.

Smith, C. and Williams, L., 1993. Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives. *IEEE Transactions on Software Engineering*, 19(7), 720-741.

Software Engineering Institute (SEIa), Available from: http://www.sei.cmu.edu/ [Accessed 14 April 2005]

Software Engineering Institute (SEIb), Available from: http://www.sei.cmu.edu/architecture/definitions.html [Accessed 14 April 2005]

Sommerville, I., 2004. *Software Engineering* 7th ed. London: Pearson/Addison-Wesley.

Song, X., 1995. A Framework for Understanding the Integration of Design Methodologies. *ACM SIGSOFT Software Engineering Notes*. 20(1). 46-54.

Sun Microsystems Enterprise JavaBeans Technology Available from: http://java.sun.com/products/ejb/ [Accessed 21 April 2005]

Sutcliffe, A., 1988. *Jackson System Development.* Prentice Hall International.

Szyperski, C., 1997. *Component Software: Beyond Object-Oriented Programming.* ACM Press/Addison-Wesley.

Tigris, n. d. Tigris.org: Open Source Software Engineering Tools, Available from: http://argouml.tigris.org/ [14 May 2005]

Udell, J., 1994. Componentware. *Byte*. 19(5).

UML Success Stories, n. d., Available from: http://www.uml.org/uml_success_stories/index.htm [Accessed 21 April 2005]

van Harmelen, M., 2001. *Object Modeling and User Interface Design*. London: Addison-Wesley.

Van Horn, R. L., 1973. Empirical Studies of Management Information Systems. *Data Base*, 5(2), 172-180.

Vliet, J. C. van, 1997. *Software Engineering: Principles and Practice (reprint)*. West Sussex, England: John Wiley & Sons.

Waring, A., 1996. *Practical Systems Thinking*. London: International Thompson Business Press

Watt, D. A. and Brown, D., 2001. *Java Collections: An Introduction to Abstract Data Types, Data Structures and Algorithms*. John Wiley and Sons.

Weaver, P. L., Lambrou, N. and Walkley, M., 1998. *Practical SSADM version 4+: A Complete Tutorial Guide 2nd ed.* London: Financial Times Pitman.

White, S. A. *Introduction to BPMN*. Available from: www.bpmn.org/Documents/Introduction%20to%20BPMN.pdf [Accessed 12 April 2005]

Wieringa, R. A Survey of Structured and Object-Oriented Software Specification Methods and Techniques. *ACM Computing Survey*, 30(4), 459 – 527.

Wilson, B., 1984. *Systems: Concepts, Methodologies, and Applications* 2nd ed. John Wiley & Sons.

Wirfs-Brock, R., Wilkerson, B. and Wiener, L., 1990. *Designing Object-Oriented Software*. London: Prentice Hall.

Wittenberg, C. H. Progress in Testing Component-based Software. In: *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis, Portland, Oregon, USA, August 21 - 24, 2000*. 178.

Wood, J. and Silver, D., 1995. *Joint Application Development 2nd ed.* New York: Wiley.

Yao, H. and Etzkorn, L. Towards a Semantic-based Approach for Software Reusable Component Classification and Retrieval. In: *Proceedings of the 42nd Annual Southeast Regional Conference, Huntsville, Alabama, USA, April 2-3, 2004*. 110-115.

Yoche, E. R. and Levine, A. J., 1989. Basic Principles of Copyright Protection for Computer Software. *Communications of the ACM*. 32(5): 544-545.

Yoche, E. R., 1989. Legal Protection for Computer Software. *Communications of the ACM*. 32(2). 169-171.

Yourdon, E., 1989. *Modern Structured Analysis*. Englewood Cliffs, N.J.: Yourdon Press/Prentice-Hall.

Zaremski, A. M. and Wing, J. M., 1997. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*. 6(4), 333-369.

Zelkowitz, M.V. and Wallace, D.R., 1998. Experimental Models for Validating Technology. *Computer*, 31(5), 23-31.

Appendix I

18.1 Glossary of Acronyms and Abbreviations

ADORA The Analysis and Description of Requirements and Architecture Method (Glinz et al 2001)

ADT Abstract Data Type

ATAM The Architecture Tradeoff Analysis Method (Kazman et al, 1998)

CASE Computer Aided Software Engineering

CBSD Component-Based Software Development

ERD Entity Relationship Diagram

FEM Functionality Entity Class Matrix

GST General Systems Theory

IM Information Model

IPI Matrix Information, Process and Interaction Matrix

IS Information System(s)

JSD The Jackson Structured Development Method (Cameron, 1989)

KobrA The Komponentenbasierte Anwendungsentwicklung or Component-Based Application Development Method (Atkinson et al, 2002)

LFD Lower-level Functionality Model

LSL Logical Screen Layout

MAP Framework Model, Architecture and Process Framework

MFD Middle-level Functionality Model

NAVITA "New component" from the Sanskrit Naviinamh "new" and GhaTaka "component"

NIMSAD Normative Information Model-based Systems Analysis and Design

OO Object-Orientation or Object-Oriented

OOSE The Object-Oriented Software Engineering Method (Jacobson et al, 1992)

Open The Object-oriented Process, Environment and Notation Method (Firesmith et al, 1997; Graham et al, 1997; Henderson-Sellers et al, 1998)

Perspective The SELECT Perspective Method (Allen and Frost, 1998)

RDB Relational Database

RSE Reuse-driven Software Engineering (Jacobson et al, 1997)

SDM System Development Method

SDP System Development Process

SSADM Structured System Analysis and Design Method

UML Unified Modelling Language

USDM User System Dialogue Model

Appendix II:

A Detailed Analytical Survey of Component-Based System Development Methods

ABSTRACT

This paper presents an extensive analytical survey of major Component-Based Software Development methods that are freely available in published books. Many Component-Based methods have their origins in popular Object-Oriented methods – the authors of which tend to fuse Object-Orientation concepts with various software reuse approaches. Their justification is that Object-Orientation itself does not deliver the promised high-level software reuse, and they argue that only by marrying various peripheral reuse concepts such as Design Patterns, Frameworks, Business Objects, Domain Engineering with Object-Orientation, the vision of assembling applications from prefabricated components may be realised. The underlying assumption is that Component-Based Software Development methods are a natural evolution from Object-Oriented methods, with a unique emphasis on reuse. In recent years, there have been accumulated interests – both academic and commercial – in Component-Based Software Development methods, evident from the growing list of publications on this topic and widespread commercial availability of component technologies. Since the late 1990s, a few system development methods claiming to be component-based have appeared in publications. This paper surveys these published CBSD methods.

Keywords: Component-Based Development, Component-Based Software Development, Component-Based Software Development Method, System Development Method, System Analysis and Design

1 INTRODUCTION

In recent years, there has been a growing trend to emphasise the need for large-scale software reuse in system development projects. One strand of this thinking is championed by a development strategy known as Component-Base Software Development (CBSD). CBSD is defined by D'Souza [4] as follows:

> An approach to software development in which all artefacts – from executable code to interface specifications, architectures, and business models and scaling from complete applications and systems down to small parts – can be built by assembling, adapting, and "wiring" together existing components into a variety of configurations.

Although CBSD has evolved from Object-Oriented (OO) methods and technologies, it is hard to ignore that there has been a major shift in thinking about the nature of software development, from which many issues arise. For instance, whilst there is a general consensus on the essential characteristics of components, there is no universally-agreed definition of the term "component". The main aim of this paper is to highlight the similarities and differences between the investigated CBSD methods.

In order to present the results of the survey in a fair and systematic way, three major features of methods – System Development Process, System Modelling and Software Architecture – will be highlighted (see Chapter 3 of the thesis).

System Modelling covers a range of important matters including modelling artefacts, concepts, and guidelines on how these models should be used to capture various aspects of the system. For example, this includes concepts of object and class, notations to represent them and techniques for producing a class diagram and its related artefacts. System Development Process deals with the issues of breaking down and

ordering the entire system development effort into stages by providing general guidance on how the development should be conducted. Software Architecture, now recognised as an important part of a method, deals with the issue of creating a good overall structure of the system so that it will be reliable, maintainable etc.

In addition, other important aspects of methods, such as presumption about the nature of component, will be considered.

The investigated methods include Reuse-driven Software Engineering [8], SELECT Perspective [1], Catalysis [4] and KobrA [2].

Discussions on each of the methods are organised in chronological order of publication. For each method, introductory information on the method background is followed by detailed discussion of Software Architecture, System Modelling and System Development Process, and a short summary. Concluding remarks on this analytical survey can be found in Section 7, and the references are listed in Section 8.

2 REUSE-DRIVEN SOFTWARE ENGINEERING (RSE)

Jacobson et al [8] discuss the concept of Reuse-driven Software Engineering (RSE). The component-based reuse approach they describe draws upon Object-Oriented Software Engineering – A Use Case Driven Approach [6], The Object Advantage: Business Process Reengineering with Object Technology [9], UML [3][7][10], the concept of domain engineering and years of experience of applying reuse strategies by the authors.

2.1 RSE: Overview

In order to make software reuse effective, Jacobson et al [8], suggest that there are four dimensions of reuse that need to be addressed. They are:

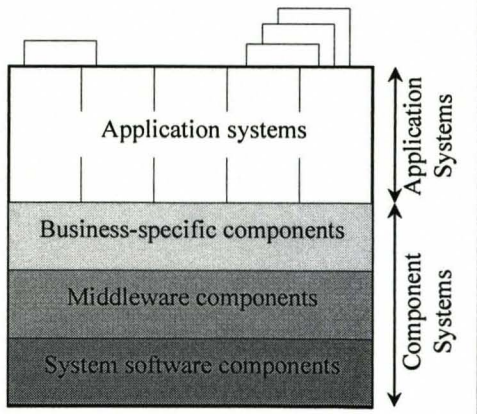
- ⇨ business orientation – concerned with the issue of ensuring that the reuse is effective both in terms of cost and time-to-market
- ⇨ engineering orientation – concerned with the methodological process of creating components and building applications out of these components
- ⇨ technical sequence – concerned with the issue of ensuring that models traceable from the highest level of abstraction down to code (since reuse at abstract modelling level is thought to be more effective than mere code reuse)
- ⇨ business process engineering - reengineering of business process as well as the system development process itself using OO concepts.

2.2 RSE: Software Architecture

RSE suggests the following model (Figure 2.1) of layered architecture for systems in which application families or groups of related applications share individual components or sets of components at all levels of implementation.

In the model, each vertical column at the upper half of the box represents a distinct application system – variants of which are shown as vertical columns in the background. For example, an application developed for use in a particular country may need to be adapted with elements specific to another country if the same application is to be used elsewhere. Those elements are variants. Applications with similar features are grouped into application system families that share components from the layers beneath. The business-specific components layer contains components that are particular to the type of businesses that use the application. Middleware components are components that offer platform-independent services to business-specific components and application systems, while system software components are components that offer Operating System, networking, hardware interface and other services.

Figure 2.1 RSE Reference Architecture



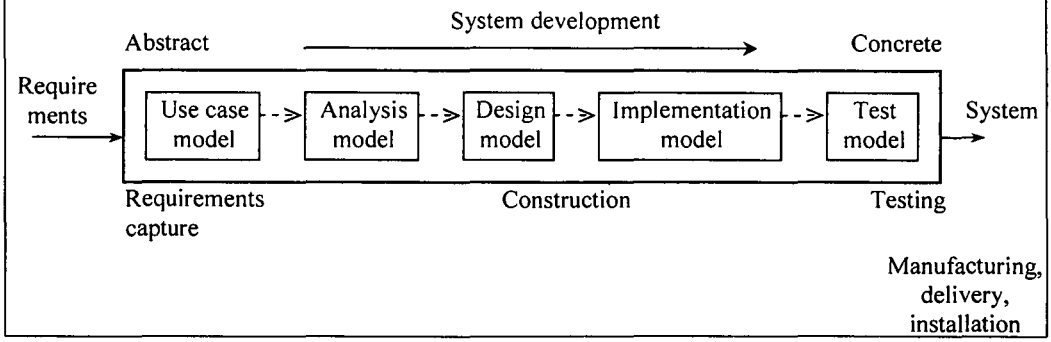
2.3 RSE: Modelling

RSE and its sister method OOSE [6] regard software engineering as a systematic model building process (Figure 2.2). Modelling in RSE starts with a "requirement capture"; implementation and testing mark the end of the technical development process. As in OOSE, RSE suggests the use of the following models and diagrams:

- ⇨ Use Case Model – Use case diagram

- ⇨ Analysis Model – Class diagram
- ⇨ Design Model – Sequence diagram
- ⇨ Implementation Model - Implementation
- ⇨ Test Model – Use cased-driven test cases

Figure 2.2 Software Engineering as a model building process



OOSE stresses the need to ensure traceability of models by taking use cases as the basis for system development, hence the name use case-driven approach. The same emphasis made in RSE. Since a number of these diagrams have now become part of our OO lingua franca, descriptions of them will be kept minimal in the subsequent sections.

It is worth considering what RSE regards as a component and how this affects modelling in this method. RSE defines component as 'anything specifically engineered to be reusable'. Therefore, any development artefact – be it a class, a use case, a fragment of a class diagram, a sequence diagram, a program, a test case, a project plan, etc – is potentially a component. In order to maximise granularity of reuse, models are packaged together. For example, in RSE, there is a clear thread of development running from use cases down to coded programs, and RSE suggests that they can be packaged together on that basis. Reusing a use case means reusing all classes analysed in the analysis model (for the use case), sequence diagram in the design model, programs in the implementation model and test cases in the test model.

One major dimension to reuse is genericity. In order to make something reusable, it is important to make it generic; the more generic an artefact is the more reusable it becomes. Therefore, RSE reminds us of a range of 'variability mechanisms' that are at the disposal of developers. These include: Inheritance for classes, Uses and Extends for use cases, Parameterisation for classes, Configuration and Module-interconnection languages and other CASE Tools related facilities. Using these mechanisms in all stages of development, RSE intends to make all artefacts more generic and reusable.

2.3.1 Use Case Model – Use Case Diagram

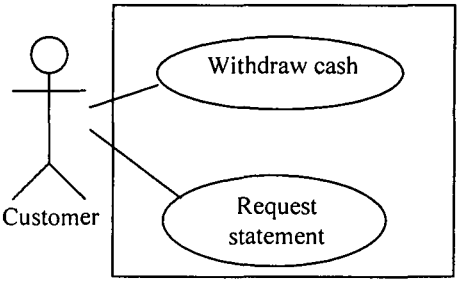
- ⇨ This model, pioneered by the same main author in [6] and later integrated into UML, has become a popular choice with developers. Use cases are primarily employed to capture the system requirements (Figure 2.3).

Main Concepts

The main concepts in this model are:

- ⇨ Actors – represent the role of those that interact with the system
- ⇨ Use cases – represent a significant sequence of transactions performed by a system, which 'yield observable results of value to an actor' [6].Use cases can have two kinds of relationship with other use cases: <<uses>> and <<extends>> which indicate common and exceptional behaviour among use cases.

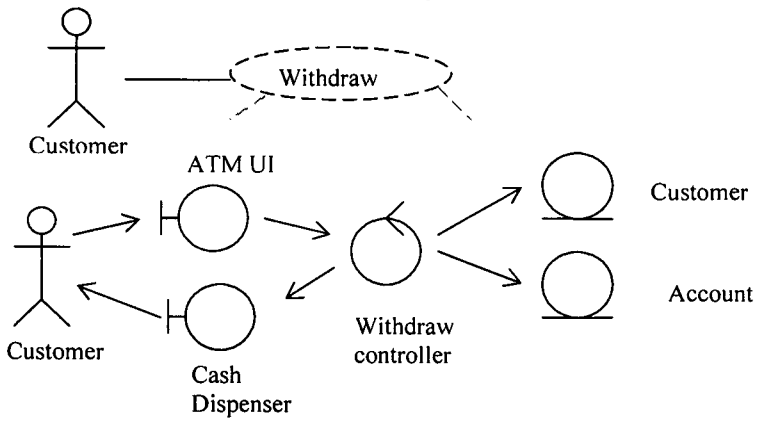
Figure 2.3 A simple use case diagram



Technique

Actors are first identified by investigating who will be using the system and the roles they would play when they use the system. Following up from this, the way these actors use the system is represented as a use case. Once the use cases are stable, <<uses>> and <<extends>> relationships are applied. Development of use case model is said to be an interactive, iterative and often creative process. Later enhancements of this model include supplementing the diagram with interface prototypes, JAD sessions etc [275].

Figure 2.4 Class Analysis in RSE



2.3.2 Analysis Model – Class Diagram

This is essentially a class analysis in which classes are identified for each use case and categorised into three kinds of stereotyped classes (Figure 2.4).

Main Concepts

- ⇨ Entity classes – have long-lived states and attributes
- ⇨ Boundary classes – interact with actors
- ⇨ Control classes – 'perform use case-specific behaviour'

Technique

Descriptions of use cases are analysed in order to uncover various entity objects that contribute to the use case. Typically one interface object is created for each actor and control objects are created as required by the collaboration.

2.3.3 Design Model – Sequence Diagram

This diagram shows how various objects in the use case – boundary, control and entity objects – pass around messages in order to realise the use case (Figure 2.5).

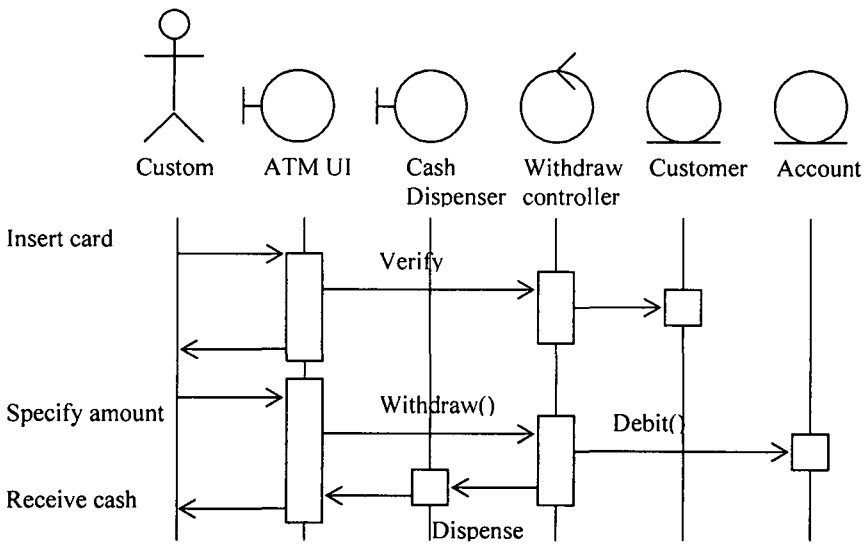
Main Concepts

The main concepts are objects and messages.

Technique

The analysis model is taken as a basis for this diagram. For each use case, the analyst would go through the description step by step, determining the messages that need to be passed between objects in order to achieve what is required in the use case.

Figure 2.5 A simplified Sequence Diagram for Withdraw Cash



2.4 RSE: Development Process

The system development process is broken down into four major sub-processes, three of which are technical processes and the other, a management one. Each of these is a model building process in its own right with different emphases in different processes (Figure 2.6).

Application Family Engineering (AFE) is a process that determines how to decompose the overall set of applications into a suite of application system and supporting component systems. An application system family is a set of application systems with common features. There are three types of application system family: a) an application system suite is a set of different application systems that are intended to work together b) application system variants are a type of application system, which need to be configured, packaged, and installed differently for different users and c) some sets of otherwise fairly independent application systems can be treated as members of a family, by building them from the same sets of lower-level reusable components.

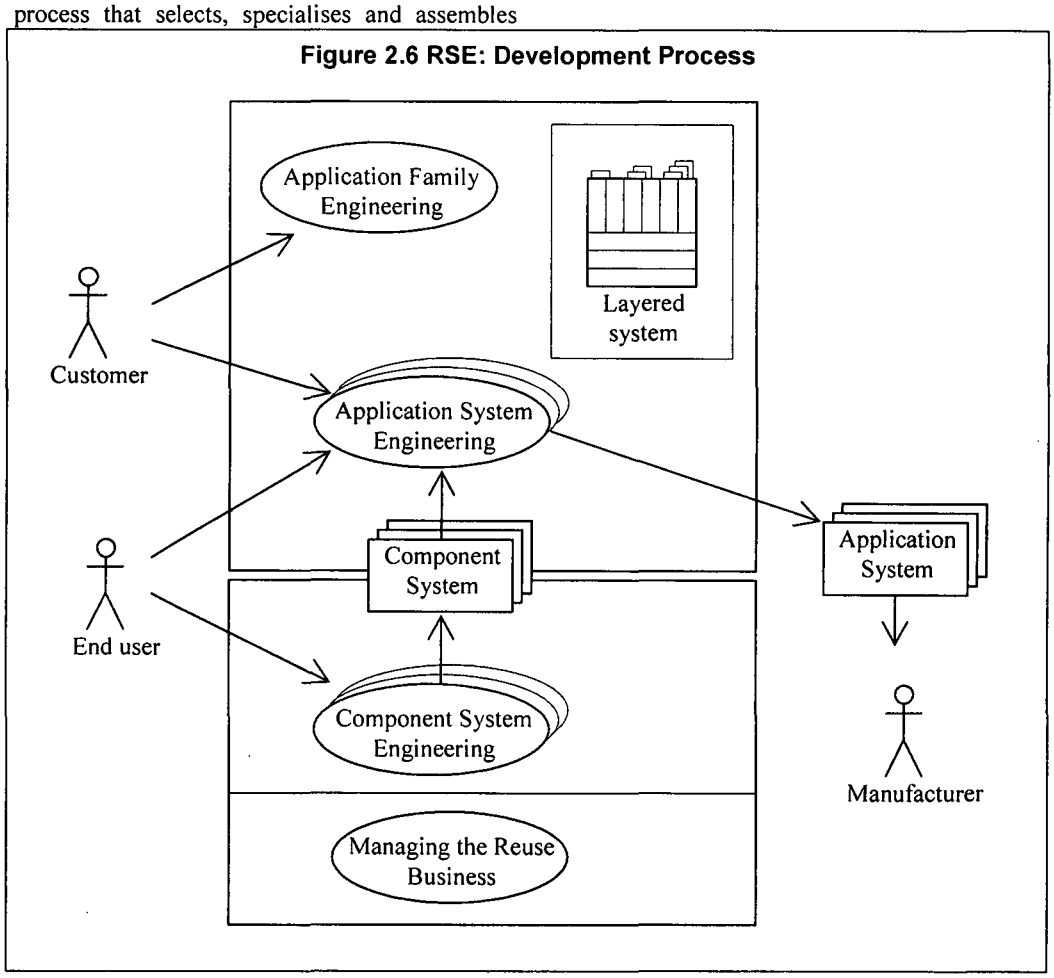
Application System Engineering (ASE) is a process that selects, specialises and assembles components from one or more component system into complete application systems. It uses appropriate tools, methods, processes, and instructions provided explicitly for the component system.

Component System Engineering (CSE) is a process that designs, constructs, and packages components into component systems. The process will use appropriate code, template, models, documents and perhaps custom tools. A component system is a set of related components that accomplishes some function larger than that accomplished by a single component. Instead of dealing with thousands of elements, a reuser can restrict his or her scope to a few hundred components, packaged into a relatively small number of component systems.

2.5 RSE: Summary

Modelling techniques of RSE are heavily borrowed from OOSE. Development process is upgraded from the OOSE counterpart in order to bring it in line with the architecture, which is simple and common. RSE has devoted the rest of its presentation to the importance of the organisational dimension for reuse business.

Figure 2.6 RSE: Development Process



3 SELECT PERSPECTIVE

Component-Based Development for Enterprise Systems: Applying The SELECT Perspective™ (or *Perspective* in short) [1] is the successor to the OO method, also called SELECT Perspective [5].

3.1 SELECT Perspective: Overview

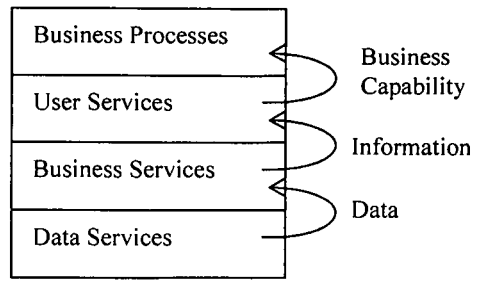
Perspective is defined as a 'collection of industrial best-practice modelling techniques that are applied and adapted using process templates within an architectural framework across a wide range of developments in a component-based setting' [1]. The principle doctrine of this method is reuse of service-oriented components (i.e. code) through tried and tested processes, architecture and modelling techniques. This method heavily borrows ideas and concepts from UML and other OO methods, including SELECT Perspective v1 [5], BPM, Patterns, the responsibility-driven approach and service technology.

3.2 SELECT Perspective: Software Architecture

The architectural model suggested by Perspective is neither radically new nor technically complex. It is a simple three-tiered architectural model, much like the classic Model-View-Control (MVC) model. What is interesting, however, is the integration of the concept 'service' into this architectural model. The term service is used in Perspective to mean a 'collection of related functionality' that can only be 'accessed through a consistent interface'. In OO terms, a service generally means a coherent group of class operations that is meaningful in business sense. Therefore, the granularity of a service is typically higher than normal class operations, coming somewhere closer to the granularity of a use case. Alternatively, services are similar to 'responsibility' of the CRC approach [12].

In order to construct higher-level objects that provide high-level operations, rather than low-level class operations, Perspective introduces control objects [6] that encapsulate groups of operations and act as a kind of interface for the services. Therefore, service classes, not ordinary classes, are the basic material of software architecture in Perspective. Control classes that provide at least one service are called service classes, and each service layer in the Perspective architectural model is made up of service classes, see Figure 3.1.

Figure 3.1 Layered architectural model of SELECT Perspective



The Data Services layer enables the Business Services in the layer above to access data in a technology-independent manner. Business Services mainly perform data transformations, and User Services allow the users to enhance their business capabilities by making use of the services offered by the system.

3.3 SELECT Perspective: System Modelling

System modelling in Perspective is preceded by Business Process modelling. Perspective emphasises the importance of Business Process modelling by highlighting the issue of keeping software solutions in sync with changes in business processes that are largely caused by technological advancements. This method attempts to tackle the issue by integrating Business Process modelling with traditional system modelling. Some Business Process modelling techniques, which Perspective acknowledges are borrowed from a proprietary method known as Catalyst, are provided to capture business requirements diagrammatically. Guidelines on how the Business Process Model can be interfaced with system models also given. Therefore, system modelling in this approach begins with BPM. Most system models are UML-compliant, but some extensions to concepts and notations are introduced together with subtle changes to existing modelling techniques.

3.3.1 Business Process Model (BPM)

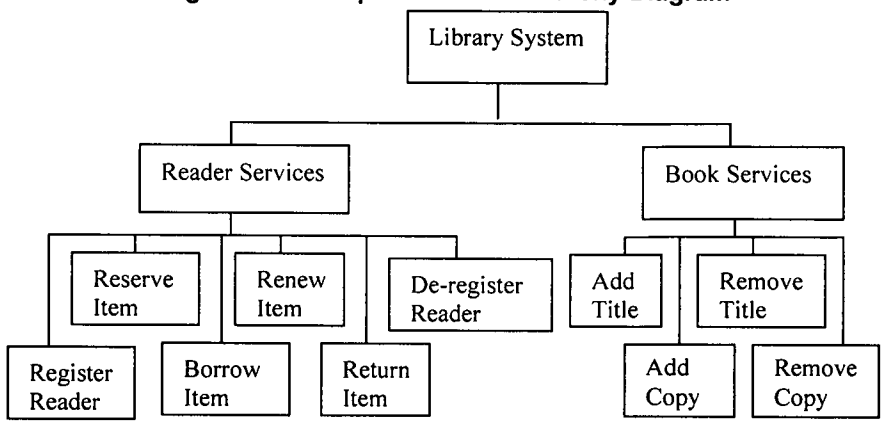
As its name implies, BPM is about modelling business activities, and is considered to be a subset of the major disciplines, Business Process Reengineering (BPR) and Business Process Improvement (BPI). In Perspective, discussions about BPM only cover the area that is relevant to modelling the process. Other important issues such as process improvement are not covered.

Main Concepts

There are two types of diagram used in this model: Process Hierarchy and Process-Thread diagrams. A Process Hierarchy diagram (Figure 3.2) shows the top-down decomposition of the

enterprise from the highest level of abstraction down to atomic business processes, known as Elementary Business Processes (EBPs). Since Process Hierarchy diagrams can only show the sequence in which the processes happen, more complex interdependencies mainly among EBPs – and often groups of EBPs – are shown in the Process-Thread diagram. Therefore, Process-Thread diagrams show non-linear dependencies among groups of EBPs – sometimes at a higher level – that are chained together by triggers.

Figure 3.2 A simple Process Hierarchy Diagram



Generally, a Process Hierarchy and some Process-Thread diagrams are first produced for the current business situation. Together these are called the As-Is model. After making improvements to the current business processes, another set of Process Hierarchy and Process-Thread diagrams are produced for the envisaged business model, labelled the To-Be model. It is important to note that what has been modelled, i.e. the business processes, is the context in which the planned system will operate, not the system itself, and only by first understanding the new context of the system, will it be possible to develop a system that fits into the business environment.

Modelling Techniques

The concept of business event features prominently in the way business processes are identified in this method. A business event is a stimulus that triggers an EBP, which could happen by the arrival of some information, a point in time, or changes in certain conditions. These events are crucial for the discovery of business processes in this approach. The approach is essentially "bottom-up" (Perspective uses the word "outside-in") rather than top-down. Based on the events and EBPs, a business process hierarchy for the existing business process is built. The non-sequential nature of EBP dependencies is modelled in the Process-Thread diagram. No detailed discussion is given as to how the processes may be improved. Changes can easily be made to the As_Is diagrams to create To_Be diagrams.

As indicated, this method provides guidelines on how to interface the business model and the system models. The general rule is that for each EBP that will be automated or partly-automated, there will be a corresponding use case. Exceptions occasionally occur when two connected EBPs are packed into a single use case, and certain individual EBPs are split into several use cases.

3.3.2 Use Case Model

Use case modelling is the first system modelling activity. The model mainly shows the functional requirements of the system (after the Business Process Improvement exercise).

Main Concepts

The model is essentially the same as in UML. The main diagram is the use case diagram that shows ways in which actors will use the system, i.e. use cases. The diagram is support by detailed textual descriptions of all the elements in the diagram. For an example of a use case diagram, see Figure 2.3.

Modelling Techniques

Although the model is very similar to the UML model, the technique used is rather different. This method proposes two 'routes' to use case modelling: Business Process Modelling, EBPs approach and Black Box System Modelling, Events approach.

Black Box System Modelling approach can be regarded as the traditional use case modelling approach, where actors are first identified and then the system's response to individual events are designated as use cases. With the BPM approach, actors and use cases are identified from the BPM. Generally, business actors become 'system actors' and for each EBP, there will be a

use case, although the relationship may not always be one-to-one. The underlying principle is that for each atomic business activity there is a corresponding system activity to support it. In this sense, a use case is regarded as a subset of an EBP.

At this stage candidate services are also identified from use cases.

3.3.3 Class Model

As in many other methods, particularly OO methods, this model plays a vital role in system modelling.

Main Concepts

Class models provide the static view of the entire system in terms its main classes, class attributes, class operations, and various types of relationships between these classes. The Perspective class diagram is same as the UML class diagram in many regards.

Modelling Techniques

This method identified two strategies for discovering classes. Many existing techniques for discovering classes fall under the category of '**business-semantics driver approach**' where grammatical analysis of business requirements specifications is performed in order to identify classes. The second approach, named '**service-driven approach**', emphasises the fact that classes must provide a set of relevant services, instead of boiling down to low-grained class operations. Class-Responsibility-Collaborator (CRC) technique [12] is the prime example of this approach.

This method advocates the use of a mixture of both strategies including text analysis, patterns, and the CRC approach.

3.3.4 Object Interaction Model

Main Concepts

Object Interaction Models show how objects collaborate in order to realise each use case. They emphasise the messages passed among the objects and the timing of those messages. There are two kinds of diagram used by these models: Sequence Diagram and Collaboration Diagram. Both Sequence and Collaboration Diagrams show the passing of different types of messages among various class instances, but the Sequence Diagram emphasises the timing of the messages passed. This method recommends the use of Sequence Diagram for the analysis of use cases and the Collaboration Diagram for the analysis of class operations.

Modelling Techniques

The strategy for creating Sequence Diagrams is to start with a small number of business objects – identified from the use case descriptions – and produce a draft sequence diagram. At the same time, a sketch Collaboration Diagram that focuses on the main scenarios and complex collaborations that cannot be shown in the Sequence Diagram is created. The initial Sequence Diagram is then revised and a snapshot of the Sequence Diagram is then depicted in another Collaboration Diagram. User Interface objects are then added. Services are also identified and reused.

3.3.5 State Model

Main Concepts

The State Model describes how states of objects with rich behaviour or complex interactions are affected by various events during their lifetimes.

Modelling Techniques

This method identifies two State Modelling approaches: the lifecycle approach for modelling behaviour of Business objects and the behaviour approach for modelling Control objects. Both approaches are event-driven. With the first approach, external events are investigated and a first-cut state diagram is then produced based on the template of birth-midlife-death events. Then detailed scenarios are brought out to enhance the diagram. In the second approach, Sequence Diagrams are investigated for Control objects with a large number of messages passed and received. The State Model is created based on these messages.

3.3.6 Component Model

Perspective focuses on modelling of business-oriented components. It suggests that there is a real benefit to be gained from reusing business objects – business services to be precise – as opposed to code from libraries. Component technology enables reuse of business services.

Main Modelling Concepts

Perspective regards components as executable code units that provide services through their published interfaces. Since Perspective views a system as layers of services in architectural terms, the concept of service is crucial to component modelling. Service is defined as a group of related operations to provide useful functionality to consumers. See Section 3.1 for further discussion. These services are grouped into physical units called service packages. Components implement these services; groups of components that support a service package are called a component package.

Modelling Techniques

Business-Oriented Component Modelling draws from a number of sources including domain knowledge, business process models, solution project feedback, generic models and patterns, legacy systems and models, legacy database and packages.

SELECT Perspective uses the following techniques:

Architectural Modelling: a high-level scoping of services into service packages, used as a reference model. For this, a) domain modelling, b) software architecture modelling, allocation of services to packages and c) reusable Business Classes (generic business classes) modelling are suggested.

Identification of services: a low-level scoping of business services to refine the previous architectural model. Services are identified either by using service types or by generic business processes.

Sowing reuse from solution projects: as the classes start to mature, candidates for future components are sowed.

Using generic models and patterns: analysis and design patters are also reused.

Perspective considers legacy assets as one of the main sources of components and provides a framework for modelling these assets as components, in particular, techniques for wrapping the legacy system.

3.3.7 Deployment Model

Allocation of software components to hardware devices is shown in this model.

3.3.8 Logical Data Model

Since objects are often stored in Relational Databases, Perspective offers a simple approach to converting objects in relational tables represented by entities of tradition logical data models.

3.4 SELECT Perspective: Development Process

Perspective suggests a dual-process approach, in which development stages are divided into two main sub-processes: the Component Process produces components out of a range of sources and the Solution Process uses components to build applications (Figure 3.3).

Figure 3.3 Duel Development Process Model of SELECT Perspective

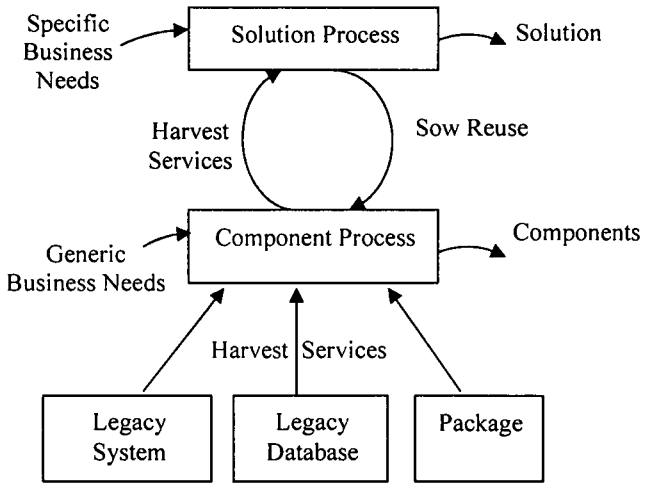
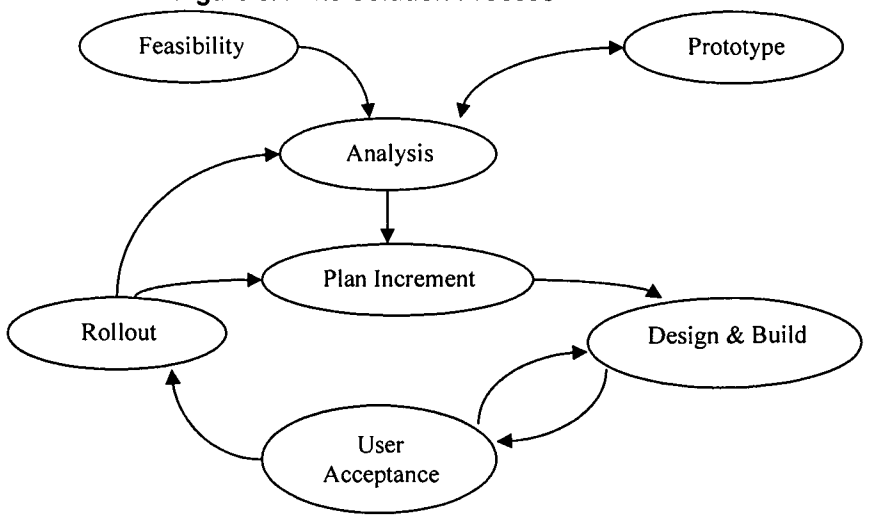


Figure 3.4 The Solution Process



Each sub-process contains an iterative, incremental and prototyped-based development model. Models for the two sub-processes are similar and they are both adapted from the OO version of Perspective.

The Solution Process stages are (Figure 3.4):

- ⇨ Feasibility: Scope the development
- ⇨ Analysis: Explore the requirements
- ⇨ Prototype: Elicit requirements
- ⇨ Plan Increment: Develop a plan
- ⇨ Design and increment: Construct, assemble, and test the software
- ⇨ User Acceptance: Ensure acceptance of an increment
- ⇨ Roll out: Install an increment

The Component Process stages are (Figure 3.5):

- ⇨ Architectural Scoping: Provide an overall context
- ⇨ Assessment: Assess needs for reusable services against the available resources
- ⇨ Plan Services: Develop a plan for a component project
- ⇨ Design and Build: Construct, assemble, and test the components
- ⇨ Acceptance: Ensure acceptance and certification of a set of components
- ⇨ Roll Out: Install a set of components

Figure 3.5 The Component Process

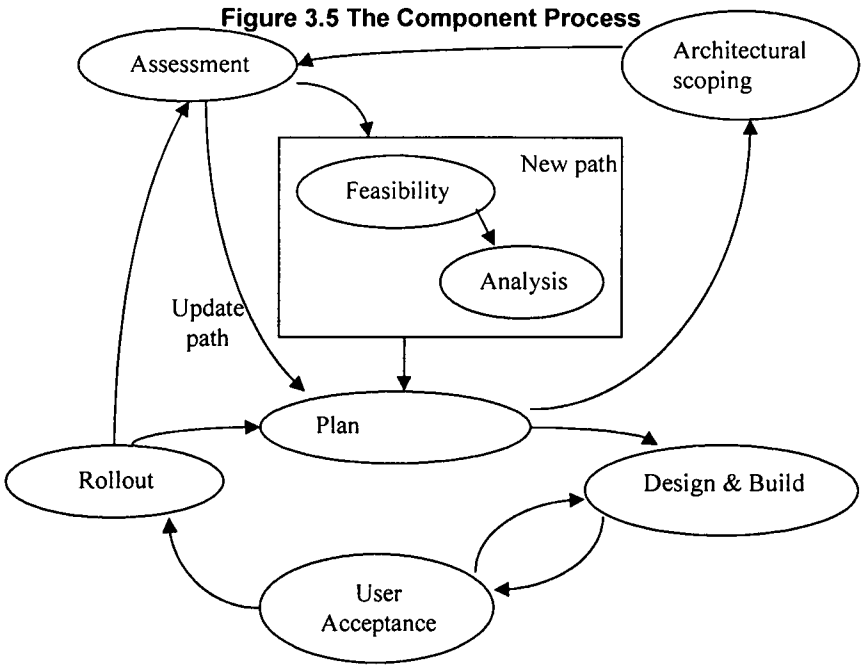
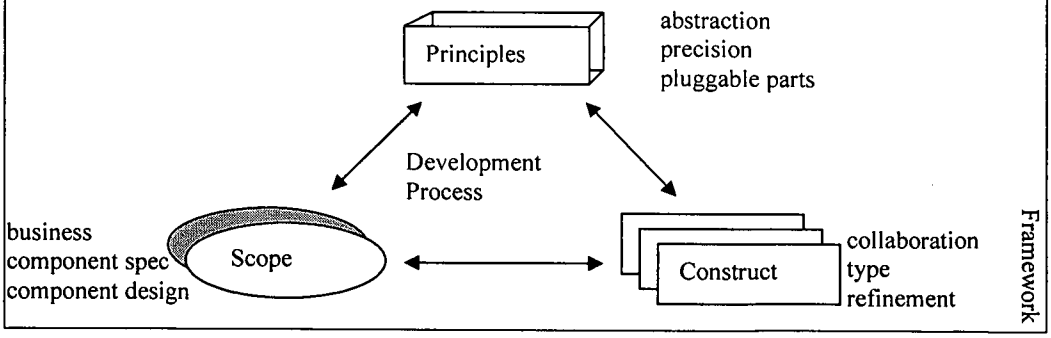


Figure 3.6 Orthogonal view of Catalysis



3.5 SELECT Perspective: Summary

In terms of models used, Perspective is mainly UML-based. However, techniques for these models draw from wider sources. Explicit inclusion of BPM in a system development method is something new. The architectural model is perhaps simplistic, while the development process seems realistic.

4 CATALYSIS

Catalysis [4] is said to be the fourth generation OO method geared towards reuse and component-based development. It draws upon a number of 'third generation' OO methods such as OMT [13], Fusion [14], Syntropy [15] as well as formal specification languages such as VDM and Z [16] and other research in this area. In terms of notation used, Catalysis is broadly UML-compliant.

4.1 Catalysis: Overview

The main features of Catalysis are summed up in Figure 3.6. Catalysis is based on the principles of abstraction, precision and pluggable parts. There are three main modelling 'constructs' or concepts: collaboration, type and refinement. The scope of modelling applies to the business domain, component specification and component design.

Modelling concepts, levels of modelling and principles are applied recursively in Catalysis. This orthogonal view of the development suggests that any model is developed for either business, component specification or component design, using either collaboration or type refinement models, and these models must comply with the principles above.

Principles

Catalysis is founded on three clear principles:

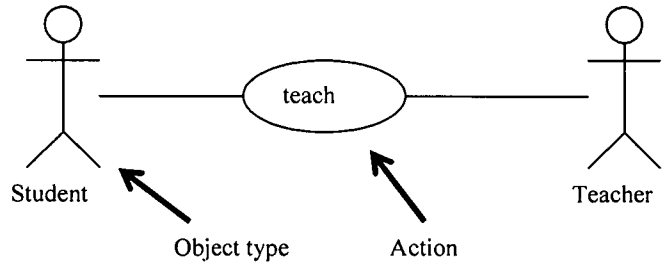
- Abstraction – this principle suggests that when dealing with complexity, we need to focus on essential aspects first. It helps us produce an 'uncluttered description of requirements and architecture.'
- Precision – this principle promotes the practice of exposing gaps and inconsistencies early on. A model can be both abstract and precise.

Modelling Constructs

Catalysis is based on the three main constructs: Collaboration, Type and Refinement.

- Collaboration – A collaboration diagram captures action that takes place between objects assuming roles relative to each other. The collaboration diagram of Catalysis is rather different from the common UML collaboration diagram. Figure 4.1 is a simple Catalysis collaboration diagram. Actions are things – represented by verbs – that happen between objects. They could be use cases as in traditional OO methods, or something lower in granularity. Actions always imply involvement of at least two objects: when modelling the business activities, it could be participating actors, when modelling the external context of the system, it could be that actor and the system itself or two objects passing messages between themselves when modelling the internal dynamics of the system.
- Type: Catalysis type is similar to the concept of class – except that the types are completely implementation-independent, while classes are not. Catalysis therefore can refer to both classes as well as components (which could contain classes or even components within it) as types.

Figure 4.1 A simple collaboration in Catalysis



- ⇨ Refinement: A relationship between abstract and more detailed descriptions of the same thing. Detailed descriptions are realisations of abstract descriptions. Refinement is applied to action and type. Actions, types and messages are therefore described in a hierarchical manner.
- ⇨ Frameworks: These are reusable model elements used throughout the development.

Three Levels of Modelling

- ⇨ Domain modelling – this modelling of the "outside" is used to understand the domain terminology, business processes, roles and collaborations. Models are created for **as-is** and **to-be** situations.
- ⇨ Component specification – this "boundary" modelling is about determining the responsibility and interface of the component (or system itself)
- ⇨ Internal design – this "inside" modelling defines the internal technical design of the system/component.

4.2 Catalysis: Software Architecture

Catalysis does not believe that there could be an architectural model to cater for the needs of all kinds of system. Hence, it suggests the use of architectural patterns, each of which puts forward a tried and tested solution for a specific type of problem. Catalysis lists some architectural patterns for different kinds of applications such as client-server applications and MVC applications.

4.3 Catalysis: System Modelling

Modelling in Catalysis is therefore all about defining and specifying collaborations and types at different levels of abstraction by reusing all kinds of generic models whenever possible. Typically, it will start with domain or business modelling using a set of collaboration and type diagrams. The system is treated as a type or component that interacts with its environment – its users or actors. The analysis of the collaboration between actors in the business domain and the system leads to the discovery of actions they jointly perform and the sub-types that interact when those actions are taking place. Those sub-types are further refined until they reach a stage where types or classes, as they are then called, can no longer be further refined. As the name implies, domain model may include aspects of the business that may or may not be implemented in the system. The main purpose of business modelling is to understand the business processes, roles, collaborations, classes and so on. The precise requirements for the new system have not been determined at this stage – which is the main purpose of Component Specification, the next level of modelling. For this, the analyst will determine the kind of actions or use cases the system will provide, and precisely define the responsibilities and interfaces of the components and the system itself. At Internal Design level, the 'how' question is dealt with by defining the way in which components of the system are to be implemented. In classical terms, it is the component/system design.

Catalysis divides its models into three: Static Models, Behaviour Models and Interaction Models. For Static Models, Catalysis discusses two main diagrams: Snapshots and Type Diagram, placing strong emphasis on capturing invariants. Snapshots, pre- and post-conditions and the State Chart are discussed for Behavioural Models and for Interaction Models collaboration and interaction diagrams are discussed.

Although Catalysis deploys UML notations, the models, concepts and techniques it uses are rather different from those used by many OO methods. Therefore, it would be appropriate to give examples of those new diagrams when necessary. The diagrams are discussed in the order they are likely to be used in a typical development project.

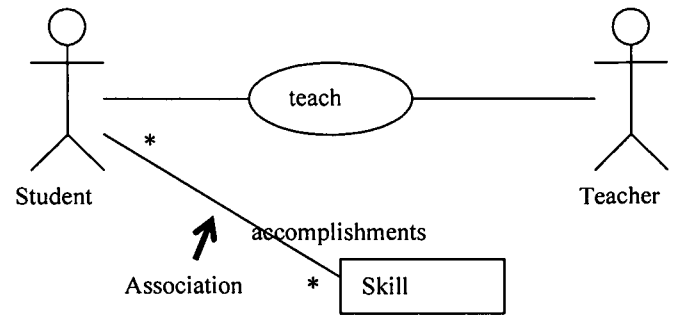
4.3.1 Collaboration Diagram

It is fair to say that collaboration diagram (Figure 4.2) is the most prominent diagram in Catalysis. Despite its name, it has little in common with collaboration diagrams of many OO methods. It has some resemblance to a class diagram with bits of use cases thrown in. It essentially defines a set of actions between objects. Since anything including actors, classes, components and the system itself is an object and actions are anything

including use case, message, event, interaction and others, one can show an infinite variety of 'collaborations' in this diagram. There is little constraint on what can or cannot be included in this diagram: some would suggest it grants a lot of expressive freedom to analysts. It is used to show the business activities, system context and the internals of the system.

Catalysis provides a rich set of concepts for this diagram; in this paper, we will examine only the key features of this diagram.

Figure 4.2 Actions in collaborations affect object state

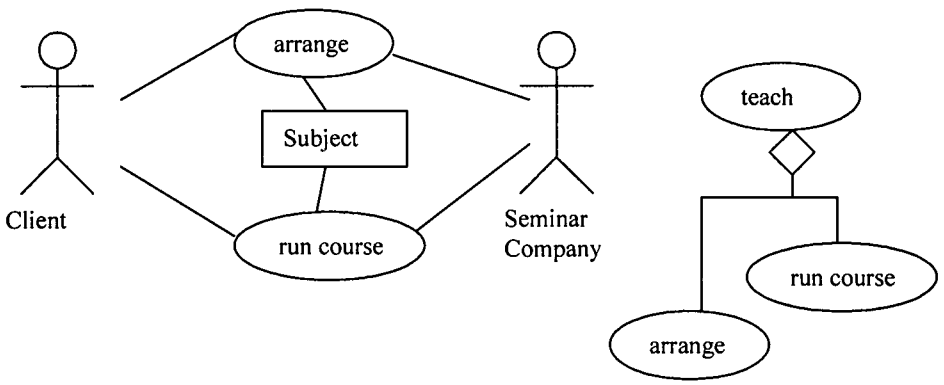


Main concepts

- ⇨ Type – see section 5.1.2.
- ⇨ Action - an action is 'anything that happens'. It is could be 'an event, task, job, message, interaction'. They are normally verbs, while classes are nouns. States of objects change as a result of actions. Therefore, effect of an action can be described in terms of state changes of one or more object/component. A group of actions that serve a common purpose is an action type, that is, a use case. In other words, a use case is composed of actions.

The diagram shows that the action "teach" involves two actors, "student" and "teacher". Actions lead to changes in state of object(s). In this case, when a student is taught successfully, the student acquires a new skill. Therefore, the class/type skill is linked to student.

Figure 4.3 Action refinement and composition



In Catalysis, actions are as important as objects. The effect of an action on objects/components is precisely described using OCL-like language (Object Constraint Language) defined by the UML. The effect of "teach", where a student gets a new skill at the end of the action, can be expressed as follows:

action (student, teacher) :: teach(skill)

post student.accomplishments = student.accomplishments@pre + skill

- ⇨ Refinement – refinement is applied to actions and types.

Refinement of Actions

Actions can be broken down into more and more detailed actions. The teach action type, in the example, can be broken down into "arrange" and "run course" (Figure 4.3).

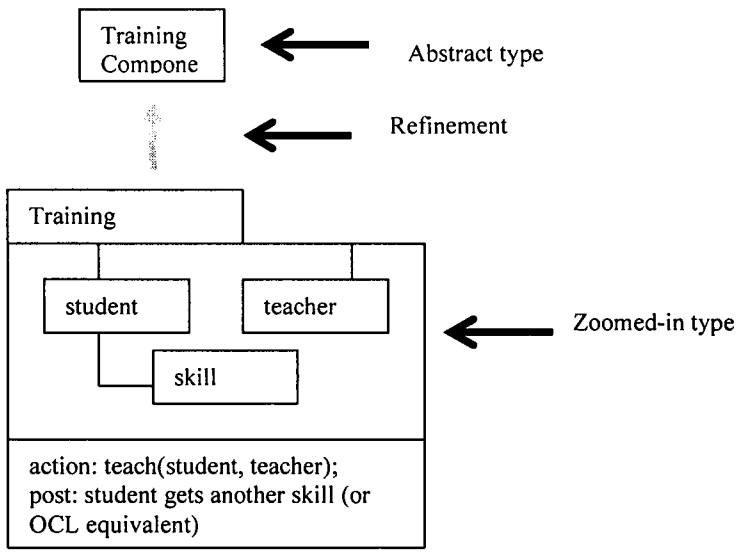
Interactions between objects are actions are shown in UML sequence diagrams. The diagram can be modified. For

instance, if there are no arrowheads in the messages, it means an action is an abstraction of more detailed interactions. See Section 5.2.2.

Refinement of Types

The refinement concept also applies to types. When types are refined, smaller types are identified (Figure 4.4).

Figure 4.4 Type abstraction and refinement



4.3.2 Interaction Diagrams

Modified versions of UML Sequence diagram and collaboration diagrams are called Interaction diagrams, the main difference being that in Catalysis, messages between objects can be aggregated into action. For example, Figure 4.5 shows a sequence diagram with aggregated messages arrange(java) and runcourse(java), which will reveal many messages when refined.

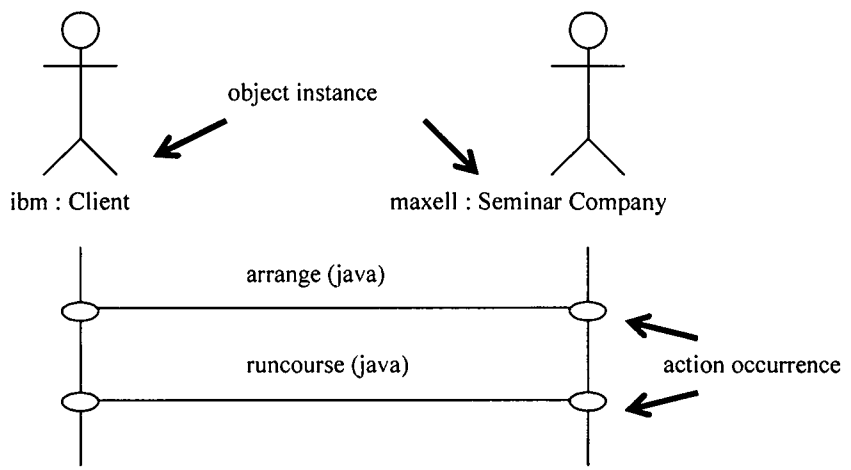
Main Concepts

The main concepts are objects, messages and actions.

Technique

This diagram is a refinement of actions that occur between objects as analysed in the collaboration diagram. Catalysis uses the recursive decomposition technique to translate actions into messages.

Figure 4.5 Catalysis sequence diagram showing actions as abstraction of messages



Techniques

Nouns in descriptions of the business domain or system indicate types, components or classes. Verbs indicate actions, use cases, messages and so on.

4.3.3 Snapshot

Figure 4.6 shows the state of a set of objects (their attribute vales and links) at a particular point in time. This diagram is used for a number of purposes such as visualising the effect of an action, exploring various business rules and constraints.

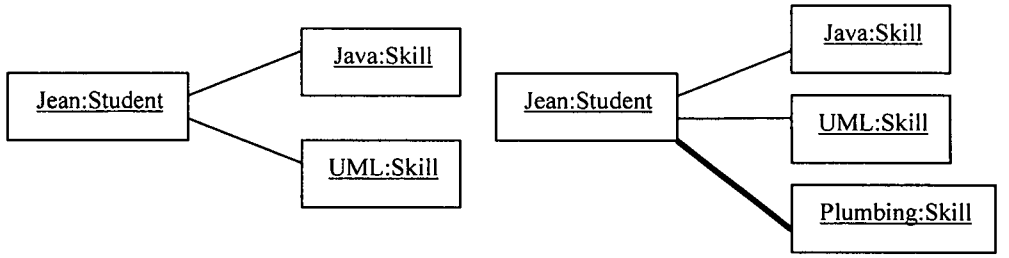
Main concepts

The main concepts used in this diagram are object, attribute and link between objects. For example, it can show the state of objects before and after a student is taught at a course, i.e. acquired a new skill. Before the student Jean attended a course, she may have two skills. On completion of a plumbing course, she would have acquired a new skill, represented by a link to the plumbing skill object.

Technique

There is no defined technique.

Figure 4.6 A snapshot showing the state of objects before and after the teach action



4.3.4 Type Diagram

This diagram is similar to the tradition class diagram. Since Catalysis is based on the principle of continuous refinement of types and actions, there may be a number of these diagrams showing various fragment of the system at various levels of abstraction.

Main Concepts

- ⇨ Type – a general term for class and component.
- ⇨ Class – same as OO classes
- ⇨ Component - components in Catalysis are not always executable code; there could be other design artefacts too. Catalysis defines component as follows.

  > Component (general) A coherent package of software artefacts that can be independently developed and delivered as a unit and that can be composed, unchanged, with other components to build something larger.
  >
  > Component (in code) A coherent package of software implementation that (a) can be independently developed and delivered, (b) has explicit and well-specified interfaces for the 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.

Components are usually represented as types in this diagram.

- ⇨ Frameworks – which are generic models describing a collaboration between some abstract classes are an integral part of Catalysis. Design patterns are also used extensively in Catalysis. Component is therefore a design concept in Catalysis. All types are potentially components.
- ⇨ Action – operations of types are often depicted as actions.

Techniques

Collaborations again provide the basis for this static model. Types identified in collaborations are continually decomposed until they are no longer decomposable.

4.3.5 State Chart

This diagram is similar to State Charts discussed by other OO methods, often under a different name such as State Transition Diagram. The main aim is to show the state dependent behaviour of a type.

Main Concepts

The main concepts are state and event.

Technique

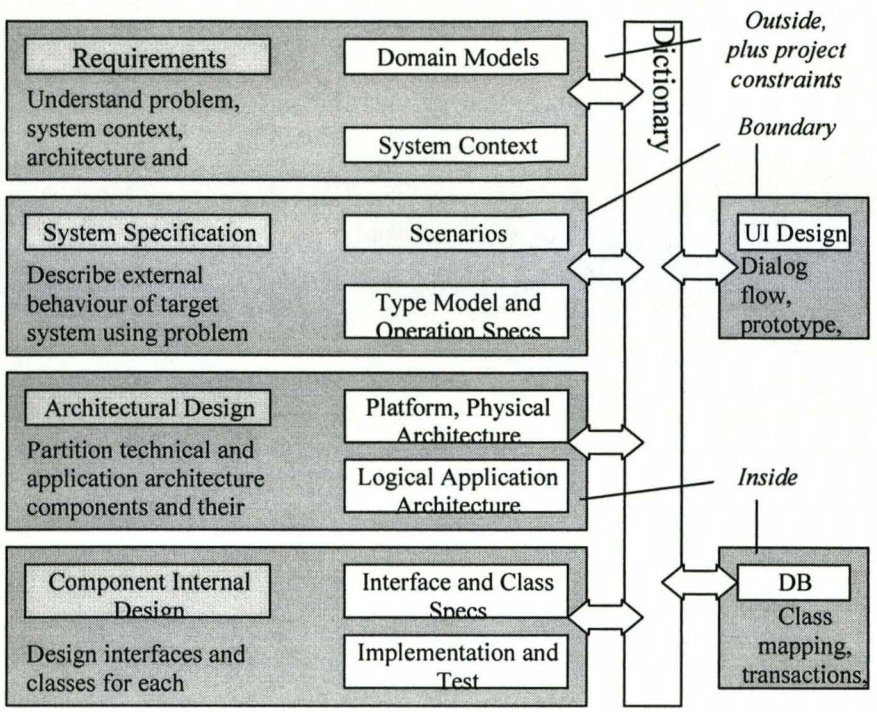
A list of states for a type is first drawn up, and actions then are examined for events. State chart is put together on the basis of the states and events.

4.4 Catalysis: Development Process

Figure 4.7 shows a very high level overview of system development as envisaged by Catalysis. It recognises that the development process is not a one-off process and that typically it is iterative and incremental. Since different development

projects requires different kinds of development process, Catalysis doesn't believe that it is possible to produce a grand and detailed process model that could be used in any type of project. This model only serves as a general reference point, and for specific projects – such as a reengineering project, component-based project and other kinds of project – Catalysis has produced so-called process patterns that are customised processes for these different projects.

Figure 4.7 Catalysis System Development Process Model



4.5 Catalysis: Summary

In terms of system modelling, Catalysis provides a wealth of innovative techniques, concepts, semantic and technical guidance, which are too extensive to be covered here thoroughly. However, it seems that there are too many loose ends in its modelling approach. Architecture and development process are not covered in detail.

5 KOBRA: KOMPONENTENBASIERTE ANWENDUNGSENTWICKLUNG ('COMPONENT-BASED APPLICATION DEVELOPMENT')

KobrA: Component-based Product Line Engineering was developed as part of a project funded by the German Federal Ministry of Education and Research.

Set against many traditional System Analysis and Design methods, KobrA is unique in one important aspect: its emphasis on Product Line Engineering. KobrA espouses a vision in which applications are developed not from scratch or even from small components, but from 'frameworks'. In KobrA, it specifically means a generic application complete with all necessary models and documentations, as well as possible variations of features. One may choose a set of variations in order to render the framework into an application. In a sense, it is the most audacious vision of reuse. It is not just about reuse of objects, or classes, or even packages of classes. Rather, it is about reusing an entire application in which very little coding or modelling is required.

5.1 KobrA: Overview

KobrA suggests that there are three orthogonal properties of development as shown in Figure 5.1.

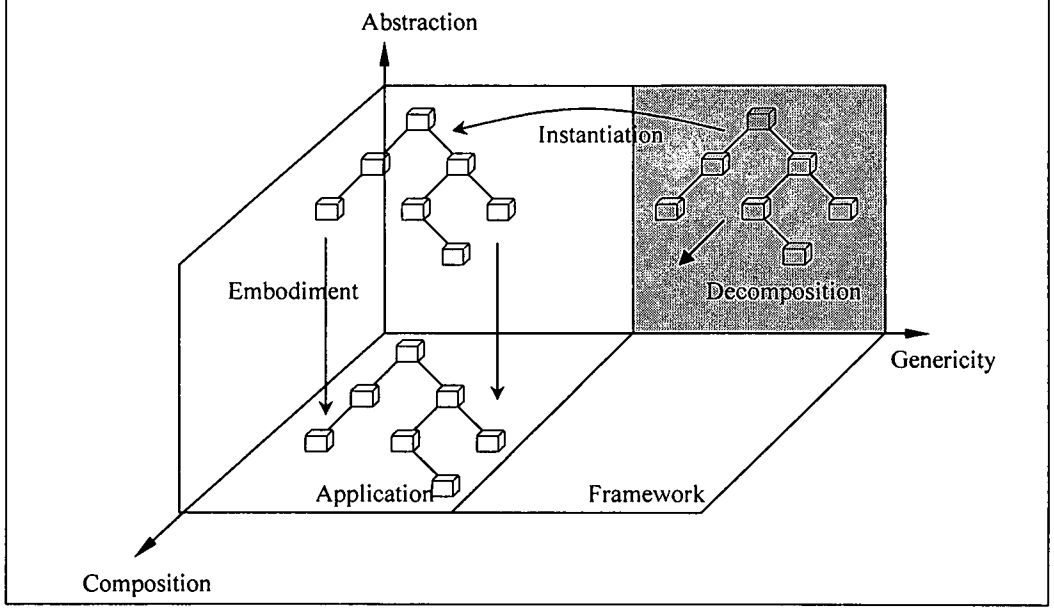
Composition/Component Modelling Dimension

Composition dimension deals with the issue of recursive decomposition of the system into finer and finer grained parts. It captures the hierarchical nature of larger components being made up of smaller ones. At the top of this hierarchy is the system – itself a component – and at the bottom are components that can no longer be decomposed meaningfully. This exercise leads to the creation of a containment tree of nested components.

Genericity/Product Line Engineering Dimension

Genericity dimension deals with the variant features of the system. At one end, there are generic applications – called frameworks - with contain elements that are common across all enterprise applications. At the other end are applications with specific features, which are adapted or extended from the frameworks. The overall development cycle is split into two parts; the first deals with the development of a framework and the second deals with the development of an application. The process of creating frameworks and applications is called Product Line Engineering.

Figure 5.1 Three orthogonal properties of system development



Abstraction/Embodiment Dimension

Abstraction dimension deals with representation of the system at different levels of detail. At a high level the system will be represented using some diagrammatic models, and at a low level, executable codes. Removing abstraction from models is called (Component) Embodiment. Embodiment is the 'act of giving a concrete form to'. The abstract models generated during component modelling are converted into executable artefacts. There are two strategies – reuse and implementation.

KobrA suggests that all component-based development projects need to deal with these three issues somehow and failure to distinguish them often leads to complexity and confusion. In this method, these three issues are addressed separately. Most importantly, developers must know, at any given point during a development project, the dimension they are working on. KobrA facilitates such a distinction.

5.2 KobrA: Software Architecture

KobrA does not suggest a reference architectural model.

5.3 KobrA: Modelling

When discussing modelling of the system, KobrA uses the terms 'Component' and 'Komponent' – it is important to differentiate between the two. 'Component' in is a general term used to describe a cohesive unit of behaviour with a commonly agreed interface. Some components are called logical components because they are represented by abstract model(s). There are also physical components – components that are executable. 'Komponent', short for "KobrA Component",

Figure 5.2 Features of a KobrA component or Komponent

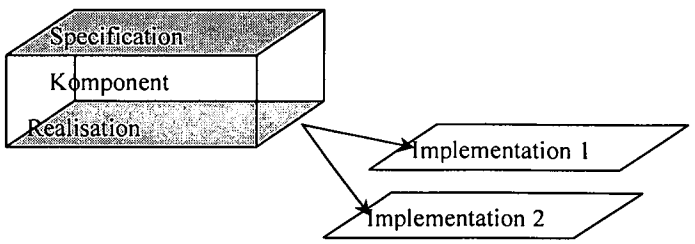


Figure 5.2 shows a component that is produced according to the KobrA approach, i.e. a component that has:

- A specification – Komponent specification defines what the component does in terms of the properties that are externally visible, dependencies it has on other components and its state-dependent behaviour.
- A realisation – Komponent realisation defines how the component is internally designed in order to satisfy the requirements expressed in the specification. Realisation of high-grained components involves identification and specification of finer-grained components. Specification and realisation of components are therefore an iterative and spiral process. The first model produced is a realisation model of the system's context – leading to specification of the system's main components. Those Komponents are then realised producing specification of finer Komponents – and so goes the process until Komponents are no longer decomposable. All Komponents have their specification and realisation expressed using models.
- An implementation – It transforms the abstract, non-executable components into executable ones. A realisation may be implemented in different ways

A range of artefacts, including models, are produced for Komponent specification and Komponent realisation.

A Komponent specification may contain up to six artefacts. They are: Structural Model, Functional Model, Behavioural Model, Data Dictionary, Quality Documentation and Decision Table. Of these, the first three are primary artefacts and the rest, auxiliary ones.

A Komponent realisation may contain up to six distinct artefacts: Structural Model, Activity Model, Interaction Model, Data Dictionary, Quality Documentation and Decision Table. Again, the first three are primary artefacts, and the rest, auxiliary.

KobrA is based on four basic modelling principles:

- **Uniformity** – every behaviour-rich entity is regarded as a Komponent, no matter what the granularity is
- **Encapsulation** - the description of what the software does is separated from how it does it
- **Locality** – all artefacts represent the properties of a Komponent from a local perspective
- **Parsimony** – every artefact should have 'just enough' information, no more or no less.

5.3.1 Structural Model

This model is used for both Komponent specification and Komponent realisation. When used for Komponent specification, the emphasis is on describing 'the externally visible types', entities or classes, in that Komponent about which interacting Komponents need to know. It must contain at least one class diagram, but for large Komponents more class diagrams can be created. When used for Komponent realisation, the emphasis is to show the refinement of the specification (class) diagram by inclusion of elements – such as attributes, operations and even embedded Komponents – that are not visible at the specification level. A structural model may be accompanied by more than one object diagram if necessary.

Main Concepts

The main concepts used in this model are borrowed from UML, which include:

- Class
- Attribute
- Operations only for realisation models
- Association
- Komponent
- Containment hierarchy

Semantics of these concepts are broadly in line with UML.

Modelling Technique

KobrA provides the guidelines for developing a structural model.

First, a draft specification class diagram is created for the Komponent. List operations of the Komponent and adorn it with the stereotype <<subject>>. Then add appropriate attributes and associations to classes. When the functional model and behavioural model are available, check them against the structural model to ensure consistency between them in terms of classes, attributes and associations.

Optionally, object diagrams can be deployed in cases of complex Komponents.

5.3.2 Functional Model

This model is used to describe 'the externally visible effects of the operations' of the Komponent in terms of what it does, rather than how it is done. It includes a set of operations specifications. Each specification can contain up to 11 parts/clauses – out of which there are two important clauses: Assume (pre-condition) and Result (post-condition).

Main Concepts

- ⇨ Operation – operation of Komponent
- ⇨ Assumes – declarative description of minimum condition that must be true to ensure successful execution of the operation
- ⇨ Result – describes the condition that becomes true after correct execution of the operation.

When producing these specifications, a number of notations from informal text to formal languages can be used depending upon the nature of the application domain.

Modelling Technique

Operations of a Komponent are identified by looking at the messages its instances receive. For each operation, an initial description of Assumes and Result clauses are developed by investigating the parameters, return data types, effects of the operation on the object(s), and appropriate assumptions. After these two clauses are developed, Receives, Returns, Reads, Changes, Sends, and Rules clauses are derived from the information gathered for the previous two clauses. The process is then repeated until intra-diagram, inter-diagram and clientship rules are satisfied.

5.3.3 Behavioural Model

This model is used to show how the Komponent behaves in response to external stimuli.

Main Concepts

It uses either UML statechart diagrams, or statechart tables. Important concepts are: **events**, **operations** and **states**.

Modelling Technique

First, choose between a statechart diagram or table. Then identify the 'externally visible logical states' of the Komponent. Identify the class attributes in the structural diagram which correspond with the state attributes. Then identify the valid operations in each state. The process is then repeated until intra-diagram, inter-diagram and clientship rules are satisfied.

5.3.4 Activity Model

Activity diagram shows a hierarchical decomposition of Komponent operations into activities. It is a flowchart-oriented view of the algorithm used to realise an operation. For each operation, participating objects are listed in swimlanes of UML activity diagram. Activities are then drawn within these swimlanes to show what each object does and the order of execution.

Main Concepts

The main concepts of UML activity diagram are: Activity, Object, Swimlanes, Sequence, Selection and Iteration.

Modelling Technique

First, the operation is broken down, as in functional decomposition, into activities. Then the flow of activities is analysed before allocating them to swimlanes based on the data types the activities use. Activities are further broken down into sub-activities (to be allocated to embedded classes). Based on the granularity of the activities, if they are 'appropriate', activities are allocated to Komponent as operations. Otherwise, further decomposition is necessary.

5.3.5 Interaction Model

This diagram reconciles the structural and activity-oriented views of the system by means of illustrating how instances interact to realise a Komponent operation. This diagram is rather similar to the activity diagram, but the focus here is the flow of messages passed from the perspective of objects.

Main Concepts

Concepts of UML collaboration (or sequence diagram, collaboration diagram being a preferred choice) diagram, object and messages are used.

Modelling Technique

Identify an initial collaboration from the hierarchy activities and objects in the activity modelling. Adjust data types, activities and allocations. Repeat this until the allocation becomes stable. Ensure that messages received by objects correspond with class operations.

Other Models and Rules

There are, of course, detailed discussions on an array of rich modelling concepts (generally UML-compliant), guidelines, and consistency checking rules. Other important issues such as project management, measuring quality attributes of models, and how to organise a large repertoire of model artefacts.

6 KOBRA: DEVELOPMENT PROCESS – PRODUCT LINE ENGINEERING

Domain Engineering lends itself as a good strategy for high-level reuse and has some influence on Product Line Engineering. Although Domain Engineering proves to be a step forward, KobrA argues that there are significant limitations. In particular, it says, Domain Engineering has problems with scoping the area of concern – it is often either too small or too large, hence it does more harm than good. KobrA argues that Domain Engineering covers areas that may or may not be affected by Information

Systems. Therefore, if the domain chosen is too small, it could fail to address important issues, and if it is too large, then it may not be cost affective. Product Line Engineering solves this problem by limiting the concerns by using the characteristics of existing, planned or future products. Everything required by the product is inside the area of concern and the rest outside it. The limitation of Product Line Engineering is that it is only useful when an organisation develops several systems in one application domain. One main task in Product Line Engineering is the identification and documentation of commonality and variability across many products or applications of the same application family.

- **Commonality**: Determination of whether a characteristic is a commonality or variability is often a strategic decision rather than an inherent property of the product family.
- **Variability**: All variables can be described in terms of alternatives. This can be difficult and involve: a) decision-making on whether a variability will be realised as a development time variability or a run-time variability, Product Line Engineering is only concerned with development time variabilities and b) representing development time variabilities at the right level of detail and presentation style
- **Decision Models**: the core role of the decision model is to show which variabilities are associated with which products. It consists of a hierarchy of decisions that relate user visible options to specific system features

Framework Engineering

In KobrA, frameworks are created using the same basic concepts, artefacts and activities as those used in development of applications, but are generalised to cover family of applications.

There are of course certain activities that are unique to this engineering, such as identification of Commonalities, Variabilities, and Komponents. Commonalities are identified using a simple scope definition table that lists various features of all members of the application family. Variabilities are identified by adopting a "product line-oriented" mind-set when introducing new elements to the model. Analysing and encapsulating the variabilities that characterise a product line provides valuable insight into consolidating functionality into good, reusable building blocks.

Application Engineering

As far as the modelling is concerned, there is not much to do. First generic artefacts in the framework are instantiated by resolving the decision models. Most of the time, the instantiated specific framework (or application) needs further adaptations. In that case, regular development techniques are to be used.

	RSE	PERSPECTIVE	CATALYSIS	KOBRA
1. Principal Doctrine	Maximising reuse through changes in organisational culture and adaptation of the development process. This is evident in SDP of RSE. The Architectural model and modelling have strong OO flavour.	Component-based (code) reuse through tried and tested SDP, architecture and modelling.	Precise specification of (model) components through rigorous modelling	Development of generic applications (or frameworks) and instantiation of frameworks to create applications through Product Line Engineering
2. Is the doctrine observable or evident in all aspects of the method?	Yes, especially the process and architecture. Modelling techniques are akin to OO modelling	The method deploys a range of popular modelling techniques. From modelling point of view, it is very influenced by OO methods. Some attempt to model components and services. The notion of assembling components is not evident	The method attempts to show that it is a rigorous modelling method founded on firm principles. There are some loose modelling techniques. As far as reuse is concerned, it mainly talks about patterns and framework, i.e. model reuse	Yes, though the applicability of this approach to different scenarios of software development is open to question
3. What is component?	Anything that is reusable – a class, a use case, a model, a test case, or a combination of any of them	Components are executable code, 'larger' than classes, and they provide services	Component is a software artefact independently developed and delivered as a unit and can be composed with other components to build something larger	A logical component is 'a cohesive unit of behaviour with a commonly agreed interface', while physical components are executable code
4. Reuse Strategy	Through internal library of components – mainly applicable to very large organisation with huge IT resources	Trough legacy system, commercial packages etc.	Patterns and framework (not like KobrA)	Frameworks
5. Modelling	UML notation but techniques are largely adapted from OOSE, the use case driven	Largely UML, both notation and techniques. Business Process Modelling is integrated. Some	Notation is largely UML, but semantics and techniques are quite different. The main concern	Notation is largely UML, but the approach is the recursive modelling of the system using

	RSE	PERSPECTIVE	CATALYSIS	KOBRA
	approach.	throw-ins of techniques such as CRC, event modelling and ER modelling.	is to identify and specify components from the outset. Top-down recursive modelling.	consistent models and techniques.
6. Support for business process modelling	Emphasis the need for business process modelling that is discussed in another book written by Jacobson – The Object Advantage.	A simple BPM described and show how it can be integrated with system modelling.	Suggest using the same models.	The need for a kind of domain modelling is mentioned, no specific techniques provided.
7. Development Process	Similar to Perspective, also include a process to deal with breakdown of sets of applications.	The duel-process highlights the nature of CBSD.	A template like minimal model. Process patterns provided for specific development scenarios.	Separation of the process into two: framework engineering and application engineering. Little further details given.
8. Architecture	A good high-level view of how application families and their composition	Simple three tier architecture.	No reference architectural model, some patterns given.	Lack architectural model.
9. Strengths	Good balance of models, architecture, development process and management	Simple and accessible	Some interesting new concepts proposed	A perfectionist approach to reuse; the only way to create an application is to instantiate a framework. Highly organised reuse is a way of life
10. Weaknesses	Notion of component is vague and modelling techniques are unexciting	More object-oriented than component-based, especially modelling. Belong to early generation of CBD methods	Effectiveness of top-down recursive modelling approach is questionable. Mainly devoted to modelling system components, patterns and frameworks. Literature not approachable	Probably not realistic. It is difficult to imagine how this approach can work with large and complex applications

7 CONCLUSIONS

This paper has systematically analysed and summarised a catalogue of CBSD methods readily available in popular literature. It is possible that there are other CBSD methods that are yet to receive wider attention. However, it is surprising that, despite its perceived significant potential, there have been relatively few published methods on this subject, in contrast with tens or possibly hundreds of object-oriented methods. It is perhaps an indication of the embryonic stage that CBSD methods are in at this moment. It could also be that CBSD is seen as a natural extension of object-orientation, rather than an overthrow of an existing way of thinking.

This survey has shown that early CBSD methods are hugely influenced by UML and other object-oriented methods. Only KobrA offers a different vision of reuse. In terms of modelling techniques, all methods surveyed use the standard OO models, use case, class, sequence, collaboration and state models. Only Catalysis offers some different modelling techniques. There has been very little technical innovation in this area. The same can be said about architecture: no CBSD method has provided fresh ideas on this issue. Only Perspective and RSE have offered detailed and realistic models of the development process.

As far as the definition of component is concerned, a standard definition is yet to be agreed. Although there is some consensus on the importance of separation of interfaces from implementation and the need to standardise them, this is not the case with executability, granularity and object-orientedness of components. In terms of emphasis on reuse, only KobrA is bold enough to make application-reuse the first priority. Other methods are fairly lukewarm about reuse at high level. They give a strong impression that systems still need to be modelled in great width and depth – which is contradictory to the notion of assembling developed components.

There are encouraging signs that dissemination of CBSD new methods and technologies is slowly gathering momentum. For example, there have been an increased number of research papers, workshops, conferences, chapters in books, and courses. Perhaps these are the final pieces falling into place for the CBSD jigsaw.

8 REFERENCES

1. Allen, P. and Frost. S., 1998. *Component-Based Development for Enterprise Systems: Applying the SELECT Perspective.* Cambridge: Cambridge University Press/SIGS Books.
2. Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wust, J. and Zettel, J., 2002. *Component-Based Product Line Engineering with UML.* Harlow: Pearson Education Ltd.
3. Booch, G., Rumbaugh, J. and Jacobson, I., 1999. *The Unified Modeling Language User Guide*, Harlow: Addison-Wesley.
4. D'Souza, D. F. and Wills, A. C., 1999. *Objects, Components, and Frameworks with UML: The Catalysis Approach.* Addison Wesley.
5. Frost S. (1995). *The Select Perspective version 4.0.* Select Software Tools White Paper.
6. Jacobson, I., Christerson, M., Jonsson, P. and Overgaard, G., 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach (revised).* ACM Press/Addison-Wesley.
7. Jacobson, I., Booch, G. and Rumbaugh, J., 1999. *The Unified Software Development Process.* Addison-Wesley.
8. Jacobson, I., Christerson, M., Jonsson, P. and Overgaard, G., 1997. *Software Reuse: Architecture, Process and Organization for Business Success.* ACM Press/Addison-Wesley.
9. Jacobson, I., Ericsson, M. and Jacobson, A., 1994. *The Object Advantage: Business Process Reengineering with Object Technology.* ACM Press/Addison-Wesley.
10. Rumbaugh, J., Jacobson, I. and Booch, G., 1999. *The Unified Modeling Language Reference Manual*, Harlow: Addison-Wesley.
11. Wood, J. and Silver, D., 1995. *Joint Application Development 2nd ed.* New York: Wiley.
12. Bellin, D. and Simone, S. S., 1997. *The CRC Card Book.* Reading, Mass: Addison-Wesley.
13. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W., 1991. *Object-Oriented Modeling and Design*, Englewood Cliffs, New Jersey: Prentice-Hall.
14. Coleman, D., Arnold, P., Bodoff, S., Dollin, D., Gilchrist, H., Hayes, F. and Jeremas, P., 1994. *Object-Oriented Development: The Fusion Method.* Prentice-Hall International.
15. Cook, S. and Daniels, J., 1994. *Designing Object Systems: Object-Oriented Modelling with Syntropy.* Prentice Hall.
16. Harry, A., 1997. *Formal Methods Fact File: Vdm and Z.* John Wiley and Sons.

Appendix III

Foundation for Rational Allocation of Class Operations

ABSTRACT

Class models are perhaps the most crucial models in Object-Oriented system development endeavours. These models, traditionally instituted at the very beginning of development stages and progressively enhanced and refined towards later stages, directly feed into implementation. Since incorrect class models can only result in unworkable solutions, the need to get class models right is crucial. As the class model evolves, operations are designated to classes in order that their objects can collaborate appropriately to realise the system functionality. Following research into the approaches for allocating operations to classes suggested by popular OO methods, it is concluded that these approaches are generally vague and ineffective, mainly because these techniques seem to assume that allocation of operations to classes is essentially an arbitrary process. An extensive series of case study-based experiments on the nature of object collaborations is conducted. Findings indicate that the allocation process may be reasoned and well-structured. This paper presents a systematisation of observations on object collaborations into a coherent set of principles providing a rigorous foundation for rational allocation of operations to classes.

Categories and Subject Descriptors

D.2.2 **[Software Engineering]**: Design Tools and Techniques – *object-oriented design methods.*

General Terms

Design, Experimentation, Languages, Theory.

Keywords

Object-Oriented Methods, Unified Modelling Language, Class Modelling, Use Case Realisation, Collaboration Diagram, Sequence Diagram.

1. INTRODUCTION

Object-oriented systems analysis using approaches such as [6] and [7] typically begins with development of a use case model to capture user requirements. It is often accompanied, or soon followed by, construction of an initial class model exploring the main business and/or application concepts. Through iterative enhancements, the use cases then reach a level of maturity at which they can be taken as a fair reflection of what the users really want. At the same time, classes, their attributes and their relationships are refined through analysis of descriptions of the requirements such as textual analysis [2], [5], and [15]. Compared to classes, attributes and relationships, operations are more difficult to identify at this stage [6]. The two models are then brought together to demonstrate how objects from the class model support or realise the functionality expressed in the use case model through a process known as "use case realisation" in UML [3], [7] and [12]. For this, analysts draw collaboration and/or sequence diagrams for each use case. Following approaches such as [6] and [7], analysts would a) identify objects from the class model which are stereotyped as entity objects, b) invent a few other objects stereotyped as boundary and control objects appropriately, and c) show how messages are passed across these objects to realise the use case.

1.1 The Problem

In the process of identifying entity objects, inventing boundary and control objects, and designating messages to these participating objects, the invention of boundary objects is relatively straightforward. In many cases, simply making up a boundary object per actor suffices. However, there are two fundamental issues that analysts have to grapple with when it comes to the other two types of objects, entity and control objects.

a) The first issue is concerned with identification of entity objects that participate in a given use case realisation. Regarding this issue, analysts can ask questions of the following kinds: -

How does one know, or determine, if an object of a class from the class diagram, i.e. an entity object, participates in a use case realisation? How does one distinguish an entity object that participates in a use case realisation from others that do not? What are the conditions or factors that influence the decision on inclusion or exclusion of an entity object from a use case realisation?

b) The second and more complex issue is generally concerned with the nature of operations allocated to participating entity objects and the use of control objects in object collaborations. The following questions characterise this issue: -

If an object is thought to participate in a use case realisation, one then has to designate operation(s) to the object. In this case, how does one determine the nature of the operation(s) to be allocated? How does one logically arrive at what an operation can do, cannot do, should do and should not do? How does one decide whether or not an object should take part in a use case realisation either actively by invoking operations from other objects, or passively by being invoked by other objects only? When is it necessary to create a control object? Other related questions also emerge from this area, but ultimately, the question is: how does one distribute intelligence fairly and rationally across all participating objects?

1.2 Organisation of paper

The rest of the paper explores answers to these questions. In the next section, the paper briefly reports on the general state of the situation regarding these issues and offer conclusions on why existing methods, to a large degree, have been less than effective in answering these questions incisively. Section 3 gives background information on the experiments we carried out which led to the synthesis of principles for allocation of class operations. Principles are discussed in Section 4 while conclusions and references can be found in Sections 5 and 6 respectively.

2. Current state of affairs

In the attempt to answer these questions, a survey of the range of suggestions, techniques and heuristics put forward by a number of object-oriented methods including the ones discussed in [1], [2], [5], [6], [7], [8], [11] and [14] has been conducted. This paper will not present summaries of each of these approaches due to limitation of space. However, general comments on these methods are offered, in particular for the two currently popular schools of thinking on this issue: the "use case-driven" approach, first discussed by Jacobson et al in [6] and the "Class, Responsibility, Collaboration (CRC) Cards, or responsibility-driven" approach, usually attributed to Beck and Cunningham, and popularised by literature such as [1], [13] and [14].

The approach Jacobson et al introduced in [6] and later integrated into the Unified Process [7] has gained increased popularity with the spread of UML. Although the use case-driven approach has popularised a number of important concepts, guidelines for allocation of operations are rather vague. For instance, Jacobson et al [6] have identified the two possible extreme situations in fair distribution of intelligence as 'fork structure' and 'staircase structure'. Their general suggestion was to use a combination of these two structures, to 'yield a stable and robust structure'. However, it seems that methodical guidelines need to be more prescriptive.

The responsibility-driven approach suggests that responsibilities of classes are identified from the requirements specification, and through role-playing, collaborations between objects are worked out. Again, guidelines given by this approach do not deal with fundamental questions such as "how does one know if a class is responsible for a certain action?"

The main weakness of these techniques is their tendency not to offer concrete guidelines on how to allocate operations. With these techniques, no object collaboration can be considered wrong, or judged to be good. Anything seems to be acceptable.

From the survey it may be concluded that a) existing methods fail to tackle two main issues (Section 1.1) rigorously and b) problems arise from basic confusion and haziness about the logical principles that underpin allocation of operations to classes.

3. Experiments – background to the principles

In order to understand the principles underlying the allocation of operations to classes, experiments have been carried out with a number of use case realisation diagrams. Two case studies, a library system and a stock management system, have been designed. For each use case in these systems, a number of use case realisation diagrams are produced by considering all possible ways in which operations could be allocated.

The implications of each style of allocation are then examined and compared with basic object-oriented principles on encapsulation, data hiding and inheritance. From these experiments, it has become clear that there are certain patterns that run through all allocations that sustain a good sense of object-orientedness, i.e. there is a good degree of encapsulation, protection of data, cohesiveness of classes and so on. The principles proposed in Section 4 are a systematisation of the knowledge, insights and experiences gained from these experiments.

Before discussing these principles, the following points should be noted:

a) The following case study scenario is used to illustrate discussions in this paper. It is a simple ordering system, named ACME Ordering System, and some use cases of the system are shown in Figure 1. It is not a complete listing of use cases in the system. More details about the system are given where necessary.

The minimal class diagram showing classes, attributes and relationships, of the system is given in Figure 2. Operations are omitted in the diagram as it is unlikely that they will be discovered at this early stage, as mentioned above.

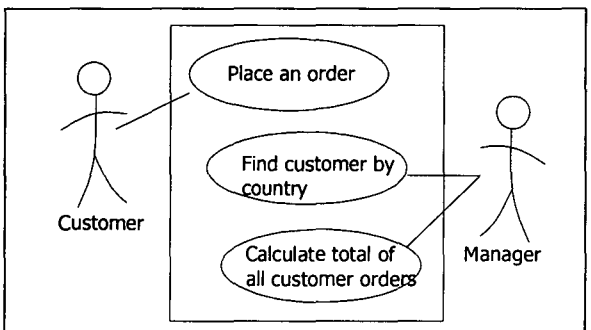


Figure 1. Use Case Diagram for ACME Ordering System

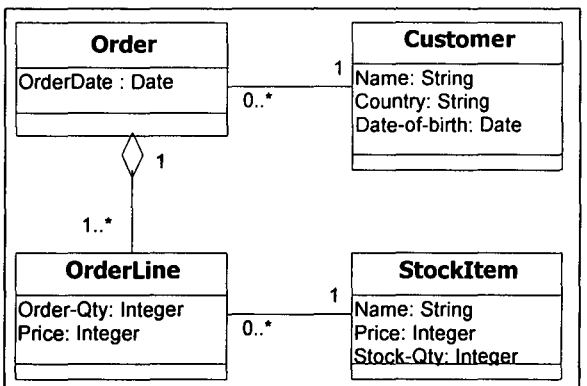


Figure 2. Initial class diagram for ACME Ordering System

b) The terminology used in this paper will refer to the terms attribute, concrete attribute, derived attribute, state, link and property. The terms are used in the following sense: object has attributes that hold data/information about the object or references to other objects. An attribute of an object can be either concrete or derived. Concrete attributes are attributes that can be thought as being persistent, e.g. Name and Date-of-birth attributes of the Customer class. Derived attributes are attributes ascertained from other, mostly concrete, attributes and can be regarded as transient. Derived attributes may be obtained from other attribute(s) of the same object. For example, Age is a derived attribute that can be calculated from the Date-of-birth attribute of Customer. Derived attributes may also be drawn from attributes of other objects too. For example, order total is an attribute that is derived from price and order quantity attributes of OrderLine objects. Most objects have link(s) to other objects through references. For example, a customer object can be linked with a number of order objects. State of an object is determined by the attributes – both concrete and derived – of the object and the link(s) the object has to other objects. For example, a stock item could be in such a state that it needs to be reordered immediately. All attributes, states and links of an object are collectively known as the properties of the object.

c) An obvious point that is worth repeating is that when an operation is allocated to a class, every instance of the class gets the operation. Therefore, if the operation OperX() is allocated to the class Customer, all instances of the class, i.e. all customer objects, can be thought of as having their own 'copy' of the operation and OperX() of each customer object may directly access its own object properties only[1]. Furthermore, since it is often useful to think at instance rather than class level, 'allocation of operation to an object' will be mentioned as well as 'allocation of operation to a class'. It follows that if an operation is allocated to an object, the same operation is allocated to other objects of the same class. Hence, the two expressions, one to help contemplate at concrete instance level, and the other at a more generic level, practically mean the same thing.

4. Principles

Observations on the general patterns that run through sensible allocations of operations are embodied in two principles. The first principle deals with the criteria for determining whether an operation can be allocated to a class and the second principle helps determine an optimal level of responsibilities across collaborating objects. The first principle is prescriptive, the second principle descriptive. These principles are generic and independent of the allocation technique that analysts may use. Any class operation technique that

[1] OMG UML Specification version 1.3 [9] does not allow operation compartment for objects to be shown. However, it is helpful often to think of objects with their own sets of operations.

sufficiently takes into account these principles can be regarded as a rigorous technique.

Although application of these principles is primarily restricted to entity and control classes/objects in this paper, there is strong evidence that are also applicable to other types of objects, including boundary classes/objects. This paper does not report on the evidence.

4.1 Principle 1: Concerning the criteria for operation allocated to a class

This principle sets out the basic conditions that must be satisfied in order for an operation to be allocated to a class.

An operation allocated to an object must access the properties of the object in order to justify the allocation. It therefore means that at least one of the following conditions needs to be met:

Criterion 1: The operation accesses the concrete attribute(s) of the object.

Criterion 2: The operation accesses the derived attribute(s) of the object.

Criterion 3: The operation accesses the states of link(s), i.e. whether or not link(s) to object(s) exist(s).

Note: Access means update or enquire.

The first criterion is simple – the operation either reads or changes the concrete attribute values of the object. In the ACME case study, allocation of an operation such as getName() to a Customer object to access the concrete attribute Name is justified. Allocating an update operation such as changeCountry(aCountry) to Customer class to change the address of individual Customer objects is also warranted (Figure 3).

The second criterion covers two cases involving derived attributes. As said, a derived attribute may depend on attribute(s) of either same object or different object(s). In both cases, if the operation accesses the derived attribute of an object, the allocation is legitimate. For example, an operation accessing the derived attribute Age in a customer object, which depends on the concrete attribute Date-of-birth, satisfies the criterion, see Figure 4.

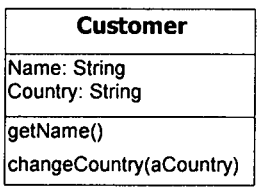


Figure 3. Operations accessing concrete attributes of a class

Likewise, Order line total is a derived value that draws from two concrete attribute values of Price and Order-Qty of the same object. Hence, allocation of an operation to calculate order line total to OrderLine class is permissible. Order total, i.e. total value of an order, is another derived attribute that depends on attributes of many other objects, namely, order line totals. Allocation of an operation that calculates order total to Order is in agreement with the criterion. See Figure 5.

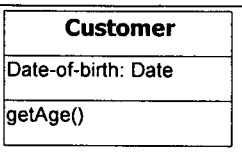


Figure 4. Operation accessing a derived attribute based on another attribute of the same class

The third criterion covers operations that change or enquire about the state of link(s), i.e. existence or non-existence of links, with other objects. For example, allocation of an operation to Customer to check the number of orders placed by a customer satisfies the criterion. It is so because such an operation will have to inspect the number of links that exist between a given customer object and any number of order objects.

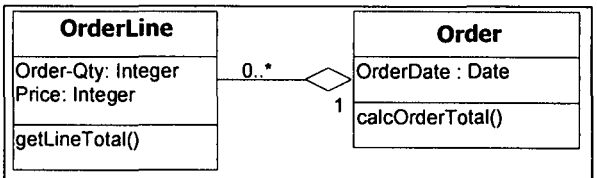


Figure 5 Operations accessing derived attributes of the same class and different class

This principle pointedly disallows allocation of operations to classes if the operation does not access properties of objects. For example, allocation of the operation Run() to the Customer class, simply because customers as persons may run when the attributes of the class are Name, Country and Date-of-birth and the relationship is to the Order class, is an incorrect allocation. None of the attributes or relationships is affected in any way by such an operation. Hence, either the operation should be eliminated or necessary attributes and/or relationships added to the class(s). The first principle, therefore, forces analysts to tie in operations with properties of the object.

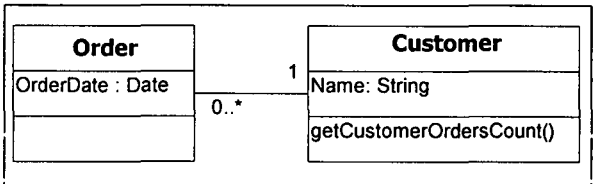


Figure 6. Operation that accesses state of links between objects

The principle also covers two other kinds of operations that all classes have: constructor and destructor. Constructor operations set initial values to attributes and effectively bring objects to 'life' and destructor operations do the reverse. Both operations affect the

properties of objects; hence allocation of constructor and destructor operations to classes is in line with the criterion.

4.1.1 Implications of Principle 1

This first principle therefore provides a rational approach to determining the involvement of entity object in object collaboration. This principle demands that an object participates in an object collaboration only if: a) the object is created or destroyed by the collaboration b) the object has its relationships with other objects established or removed c) attribute values of the object are set or read and existence or non-existence of objects and relationships are checked. If no known properties of the object are affected, then questions can be raised about missing elements in the object's properties and the relevance of the operation for that object.

An important assumption this principle makes is that only (perhaps normalised) attributes and relationships of classes provide a stable foundation for determining the allocation of class operations. Whilst some analysts may cast doubt on this assumption, it can be argued that it is a logically sound assumption totally compliant with other fundamental object-oriented principles.

The general suggestion against this assumption is the proposition that class operations or responsibilities or behaviour should be determined first because operations are innate to classes. Once responsibilities of classes are established, class attributes can be identified easily. It is often emphasised that the exact formulation of attributes is unnecessary, at least at the analysis stage, because the internals of the objects are hidden away.

Let us first discuss the point that operations are innate to classes. The point is used to imply that operations are not invented out of necessity for the system functionality but rather appear naturally when classes are conceived.

The logical conclusion of the point that there is no need to assign operations to classes depending on the functionality of the given system seems counter-intuitive. It is conceded that there are highly specialised objects, typically found in embedded and control systems, that are ubiquitous and with sets of often simple and distinct states. For example, it is obvious from mundane experience that a simple switch, unless a highly unusual one, can be turned on and off (by someone or something). When the switch is turned on, it will be on, and when turned off, it will be off, in normal circumstances. States and behaviour of such objects are so obvious that when a switch object is mentioned, most analysts rarely think of other states and behaviour for that object. To describe the behaviour of such object as innate is agreeable in a sense that these states and behaviour are not laboured out after a lengthy intellectual exercise. This is the farthest reach of the argument. To expect multi-purpose objects, such as entity objects to have operations that are so obvious that investigation of the functionality of the system becomes unnecessary underestimates this issue. Let us think of a customer object, very common among business objects, and examine what is innate to it. At an abstract or generic level, one may say customers, as the name suggests, would buy or somehow use products or services provided by the business, for which they would probably pay. However, there is nothing concrete enough to be translated into class operations. A customer of a bank may withdraw cash from her account held by the bank, and a customer of a coffee shop may not. A customer of a bookshop may purchase a book, but a customer of a library may not. It is clear therefore that if analysts do not know the business domain of the object, little can be taken as innate for a customer object, totally unlike the switch object. Even a customer object of a particular bank may have behaviour that is not applicable to a customer object of another bank, where different banks offer different services to their customers. Even within a bank, a customer may behave differently if the bank decides to offer new services or discontinue services offered in the past. The point is that the concept of customer is ubiquitous across many businesses; to claim that there is something of significance inherent to any customer object anywhere is totally unfounded. Therefore, the only way to determine the operations of any customer object is to investigate the business context of the object. This is not a straightforward exercise and requires careful thinking.

Let us now turn to the argument that class operations or behaviour are more important than their attributes. An object, by its very definition, is an entity of attributes with operations manipulating them [3], [6], [7], [9], [11] and [12]. Objects have states, attributes and links that are queried and changed through operations of the objects. Properties of an object are the existential cause for an operation being assigned to the object. In other words, it is to query or change properties of objects that operations are allocated, not that states or attributes are created to justify the allocation of operations. This argument, therefore, seems to put the cart before the horse.

Suppose that objects essentially are behavioural units that need not have a cohesive cycle of states. Objects will become stateless behavioural units without state-dependent behaviours. These objects do not aspire to the typical nature of systems and this is certainly not a vision of systems promoted by the object-oriented paradigm. The presumption that the behaviour of objects can be determined without sound knowledge about their states, attributes and links is evident in specialist objects with universally understood discrete

states. However, the claim that this concept can be effortlessly transported to the world of multi-purpose business objects is unfounded.

In fact, the object-oriented paradigm broadly favours the property-centred allocation of operations. The principle of information hiding, for example, is all about how properties of objects should not be directly exposed to other objects but rather accessed through their appropriate operations revealing as few attributes as possible. This principle implies that states or attributes exist before operations are designated to protect them. All this supports the basic assumption of this paper that when modelling classes, attributes serve as firm ground upon which the rest of development of the class model can be succeeded. Moreover, if Relational Data Analysis principles are properly followed, there can be few or no ambiguities about the placement of attributes in classes [4] and [10]. It is therefore sensible that the part of an object that can be defined relatively easily and that provides a good basis for the rest analysis should be analysed first.

However, this does not mean that behaviour should be regarded as irrelevant when modelling classes. Indeed, the overall system behaviour ultimately determines states of the objects. It simply does not follow that behaviour of classes can be determined first. Attributes and links first need to be broad-brushed, then refined as operations are allocated, as advocated in [6].

Operations of classes can conform to the first principle and yet be in the spirit of object-orientation. For example, by way of making the system object-oriented, analysts may simply allocate 'get' and 'set' operations for all the attributes in the 'classes'. If an object-oriented system is a system of collaborating objects passing messages amongst themselves, it is hard to imagine how one could espouse this vision by simply depriving important objects of any intelligence. If it is indeed true that using dumb entity objects is acceptable, which is essentially what the Structured approach suggests, the need for the whole object-oriented paradigm must be challenged. This paper acknowledges the pragmatists' argument that because classes in the end are mostly implemented as relational tables due to a number of reasons – including lack of popular object or object-oriented databases and, simplicity and ubiquity of relational databases – there is no apparent benefit in allocating intelligent operations to entity classes. In this case, it would seem most sensible to abandon class modelling completely and switch to Entity Relationship Modelling. It is beyond the scope of this research to consider the interesting issue of usefulness of class modelling over Entity Relationship Modelling and vice versa. However, this paper emphasises that where there is a need to develop a class model in the genuine and full sense of the term, analysts have to create a class model that takes into account the issue of spreading system intelligence fairly across all objects. Principle 2 concerns itself with this issue.

4.1.2 Principle 2: Concerning the criteria for fair/rational distribution of functionality

Achieving a fair distribution of functionality over classes is important for ensuring that classes have desirable quality attributes such as stability, reusability, maintainability, extendibility and so on [1], [2] and [6]. Many authors on OO methods speak of the fair distribution as an aesthetic quality – and indeed it is an intellectually pleasing characteristic. Moreover, there is a strong logical rationale behind such distribution. In this paper, three main characteristics of such allocations are suggested.

Operations of classes have achieved fair distribution of operations if all of the following criteria are satisfied:

Criterion 1: The operation allocated to an object does not defer what can be done in the object to another object. What can or cannot be done by an operation of an object, or how much an operation can do, is ascertained from the properties of the object. The amount of work an operation performs must not transcend the properties of the object.

Criterion 2: Every operation call between two objects follows the static associative relationship that exists between the class or classes of the two objects.

Criterion 3: Control – the task of calling operations – is distributed in such a way that it reflects the chains of properties that exist among objects. For each chain of properties, there has to be an object that serves as the starting point of the chain. If there is none, creation of a control object is necessary.

The first criterion defines that properties of an object ultimately determine the maximum and minimum scope of an operation that could be assigned to it. An operation is over-responsible if it is entrusted to perform things that go beyond what can reasonably be supported by properties of the object. An operation is under-responsible if it is not entrusted to perform things that could be done with support of its properties. In a complex object collaboration, over-responsibility in one object consequently leads to under-responsibility in others. Only if all objects are neither over- nor under-responsible, can there be a fair distribution of operations.

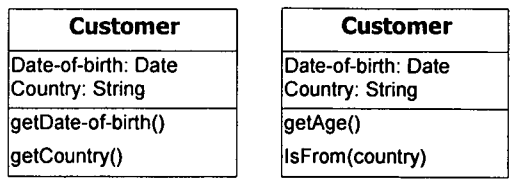


Figure 7. Operations that defer what can be done within the object to callers and operations that do not

There are some examples to demonstrate this criterion, starting with a simple operation that needs to calculate the age of a customer mentioned previously. In this

case, analysts could allocate an operation that retrieves the Date-of-birth attribute and let the caller object work out the age; alternatively, analysts could allocate an operation that carries out the calculation of age itself, as shown in Figure 6.

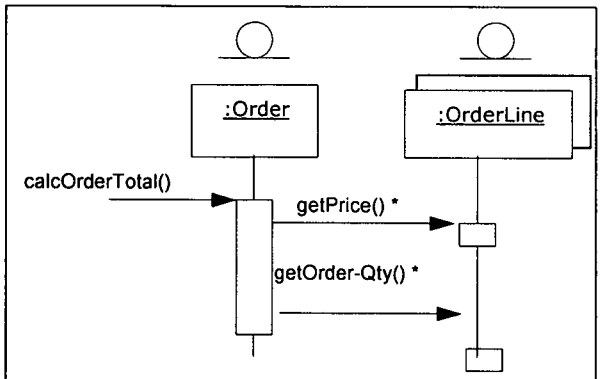


Figure 8. OrderLine objects deferring what they can do to Order objects

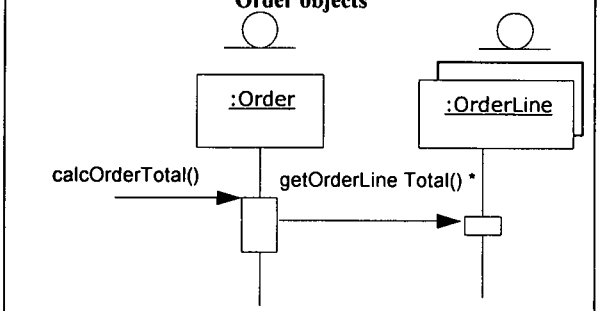


Figure 9. OrderLine objects deferring what they can do to Order objects

It is clear that calculation of age based on its own attribute is something a customer object can do itself; hence there is no need to delegate the task to other objects, according to the first criterion. Therefore, allocation of the operation getAge() is preferable. Similarly, if there is a need to check whether a customer is from a given country, analysts could either allocate an operation that retrieves the country attribute and let the caller object make the decision or allocate an operation that takes in the country as a parameter and tells the caller object if the customer is from that given country. Again, the amount of intelligence attached to the second operation is appropriate from the point of view of the properties of the customer object. It is a simple point, but in a complex collaboration, it is highly significant.

In the next example, when calculating order total, analysts could allocate operations to OrderLine objects which simply retrieve Price and Order-Qty from order line objects (Figure 7). Alternatively, analysts can allocate operations that calculate order line totals in order line objects, where only the order total is calculated in the order object (Figure 8).

Again, calculating order line totals is something order line objects can do. Delegating that calculation to the order object would wreck the balance of responsibility between objects of the two classes.

The final example for this criterion highlights a common mistake that analysts tend to make when allocating operations. If there is a need to find a customer by country, then there will be a tendency to allocate the operation to the customer class.

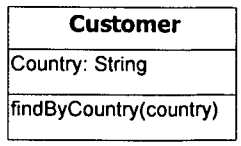


Figure 10. An over-responsible operation

This is a good example of making objects over-responsible. This allocation implies that each customer would have to know the countries that all other customers live in, i.e. access to country attribute values of all other customer objects, and find the interested ones. A customer object can only know its own country attribute value; therefore attributes of the customer object cannot support such an operation. This example will be revisited shortly.

In essence, this criterion emphasises the need to tie in operations with object properties and limit unnecessary exposure of object properties. It is a meticulous reinforcement of the object-oriented principle on data hiding.

The second criterion requires that if an object is to call directly an operation of another object, in the sense that there will be no other agent between the two objects, the caller object must know the identity of the receiving object [7]. For an entity object to know the identity of another business object, the appropriate class or classes of the objects need(s) to have a static, usually bi-directional, relationship. Operation calls among entity objects therefore must follow the static relationships identified among the classes of those objects. If two entity classes do not have a static relationship, objects of the classes cannot make operation calls to each other. Therefore, for the class diagram shown in Figure 2, only the following basic operation calls are permitted between entity objects. Let us assume that x —▶ y represents an operation call from object x to y.

:Order —▶ :Customer, :Customer —▶ :Order,

:Order —▶ :OrderLine, :OrderLine —▶ :Order,

:OrderLine —▶ :StockItem, and :StockItem —▶ :OrderLine.

Multiplicities of relationships must also be taken into account. If a number of orders can be placed with a customer, as indicated in the class diagram, a customer object may call operations from a number of order objects. On the other hand, since there is only one

customer for an order, an order object may call operations of only one customer object.

These basic operations can be concatenated. For example: -

:Customer —▶ :Order —▶ :OrderLine

:Order —▶ :OrderLine —▶ :StockItem

According to this criterion, the following operation calls are invalid given the relationships in the class diagram in Figure 2.

:Customer —▶ OrderLine, OrderLine —▶ :Customer

:Customer —▶ StockItem, StockItem —▶ :Customer

:Order —▶ :StockItem and :StockItem —▶ :Order

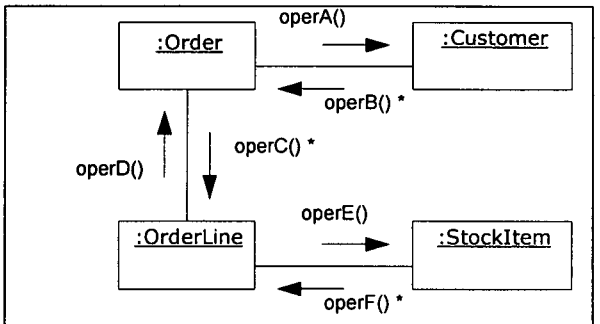


Figure 11. All possible operation calls between entity objects of ACME Ordering System which satisfy the second criterion

The validity of operation calls is better seen in collaboration diagrams that show links between objects, where it can easily and visually be compared against the class diagram. The following collaboration shows all possible valid operation calls among the entity objects in ordering system.

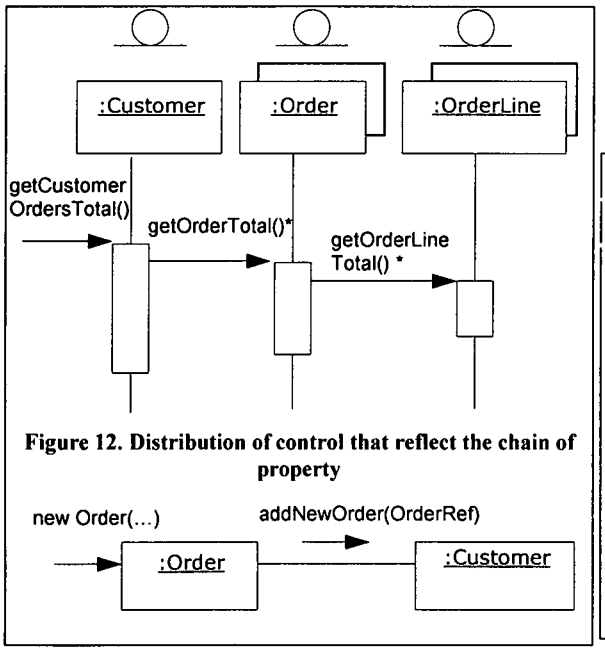


Figure 12. Distribution of control that reflect the chain of property

Figure 13. Distribution of allocation mirroring the dependency of properties of objects

In the previous example of allocation, an operation to the Customer class to find customers from a given country would require a static link between the customer object and the rest of the customer objects in the class. According to this criterion, there would have to be self-referring relationship with the 1 and 1..* multiplicities in the Customer class. If such a relationship existed, this would have been mechanically possible. However, such allocation is not desirable from the point of view of responsibility.

When more than one entity object participates in a use case, it becomes necessary to determine which object or objects will call operations of other objects. This is called distribution of control and the third criterion helps determine the way in which the distribution should be performed.

Objects, as mentioned, have sets of attributes, links and states, which are collectively known as properties. Properties of objects are connected to each other (through links) in the way that accessing properties of an object often requires accessing properties of related or linked objects. Here, they are referred to as "property chains". The third criterion suggests that the distribution of control must reflect the dependencies in the property chains.

For example, accessing total value of an order is dependent on prices and quantities of many order lines related to that order object. Therefore, the operation that calculates the order total must call the operations from order line objects to access required attributes. Allocation of operations in Figure 9 satisfies the third criterion of this principle.

Similarly, calculating total value of all orders placed by a customer is dependent on the calculation of the total value of each order, which is in turn dependent on the calculation of the total value of each order line of the order.

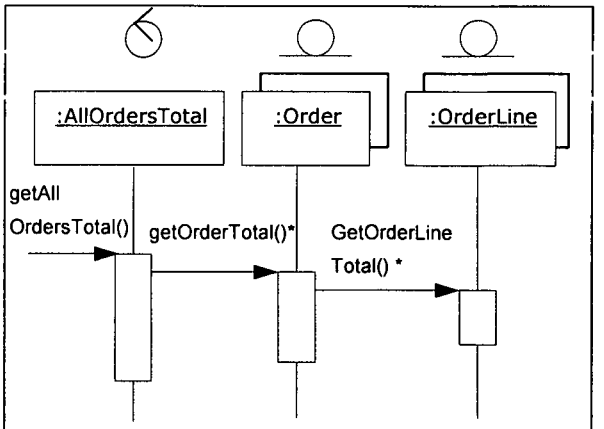


Figure 14. Control object that acts as the starting point for the property chain

Similarly, adding a new order object for a customer requires updating the customer object's links to order objects. Therefore, the operation that creates a new order object, i.e. the constructor operation, must call an operation from the customer object to add a reference to the new customer object.

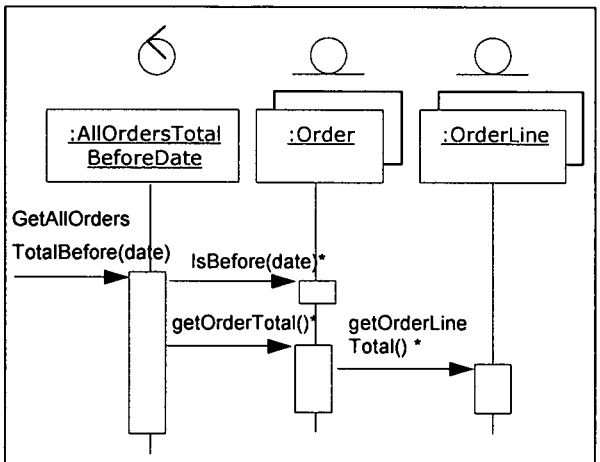


Figure 15. Criteria of principles are applied pervasively

In the previous examples, chains of property converge at a single value, usually a derived attribute of an entity object. The total value of a single order, the total value of all orders of a customer and the reference to a new order object in a customer object, are all single values that can be attributed to individual entity objects. Often, these single values do not necessarily belong to any known entity object. For example, consider the total value of all orders placed by all customers. In this case, the total value does not belong to any entity object as a derived attribute. In such circumstances, the principle suggests control objects need to be created.

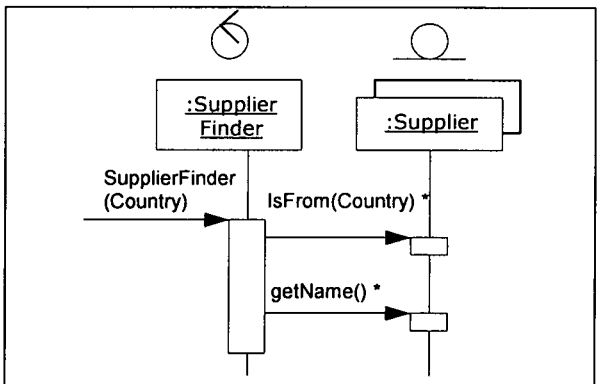


Figure 16. Control objects can be used when access to random groups of entity objects, which cannot be accessed through existing links among entity objects, is required

In this case, the control object serves as an object that holds a derived attribute or a property that does not belong to any known entity object. To be in line with the second criterion of Principle 2, the class of this object will have to have a 1 to 1..* relationship with the Order class.

If one is only interested in calculating the total value of all orders placed before a given date, since no known entity object is thought to have such a property, analysts must also invent another control object. This new control object has to filter out orders that are not placed before the given date. To do this, analysts can retrieve the date from each order object and determine whether the order was placed before the given date. Alternatively, as discussed in Section 4.1.1, analysts can pass the date as a parameter to order objects and see if they are pre-dated. Where this is the case, another message can be passed to retrieve the total value from the order object.

Finally, to find customers from a given country, an object that holds references to all customer objects is required. There is no such entity object in existence; hence, it is necessary to create a control object (Figure 16).

Overall, the second principle ensures that operations of an object do not carry out tasks beyond the object's properties.

4.1.3 Inheritance

The concept of class inheritance plays a crucial role in object-oriented analysis and design. The principles discussed in this paper are largely concerned with the distribution of operations over classes of similar abstraction (i.e. low abstraction), and do not specifically discuss the distribution of operations over generalised and specialised classes. There are two possible scenarios regarding inheritance. In the first scenario, the specialised classes have their own distinct attributes. It is clear from the two principles that any operation that requires access to the attribute in the specialised class has to be allocated to the specialised class. Otherwise, it is likely to be assigned to the generalised class.

In the second scenario, the specialised classes may not have their own distinct attributes, i.e. all the attributes they have are the same as the generalised class. In this case, the principles discussed here will not apply. However this limitation of the principles is not a major problem. Having at least two classes with exactly the same attributes that behave differently means that analysts are certain about the purpose of each of the classes. Therefore, if there is an operation that should be allocated to one of many such classes, it would be very obvious to analysts where the operation belongs.

5. Conclusion

It is fair to suggest that the class model is the single most important model in development of most object-oriented systems. Despite its overwhelming importance, class models are treated in a relatively lax manner by object-oriented methods. This is particularly true when applied to allocation of operations to classes. Techniques, guidelines and other forms of instructions

are often not instructive. The basis of the problem, it seems, is that there are no hard and clear rules based on which allocations of operations can be carried out. There are, however, general principles of object-oriented paradigm that govern class models. Clear interpretation and conscious application of those principles in object collaborations are required. The experiments here uncover specific principles that underpin object collaborations espousing the vision of object-orientation. The first principle deals with the issue of determining participation of objects in an object collaboration, and the second more prescriptive principle deals with the rational basis for distributing intelligence fairly. The principles are demonstrated in this paper mainly by using entity and control objects from a small case study. When criteria of the two principles are applied, allocation of operations becomes a rational process; allocated operations are intelligent, fairly distributed and aesthetically elegant. This gives rise to the wider hypothesis that these principles are universal, applicable to any type of object including interface objects. This is a matter for further research.

6. REFERENCES

[1] Bellin D. and Simone S. S. *The CRC Card Book.* Addison-Wesley, Reading, Mass, 1997.

[2] Bennett S., McRobb S. and Farmer R. *Object-oriented systems analysis and design using UML.* McGraw-Hill, London, 2002.

[3] Booch G., Rumbaugh J., and Jacobson I., *The Unified Modeling Language User Guide.* Addison-Wesley, Reading, Mass, 1999.

[4] Date C. J. *An Introduction to Database Systems* (7th Edition). Addison Wesley Longman, Inc., 2000.

[5] Derr, K. W., Applying OMT: A Practical Step-by-step Guide to Using the Object Modeling Technique, SIGS Books, New York, 1994.

[6] Jacobson I. et al. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, Reading, Mass, 1993.

[7] Jacobson I., Booch G., Rumbaugh J., *The Unified Software Development Process.* Addison-Wesley, Reading, Mass, 1999.

[8] Maciaszek L. A. *Requirements analysis and system design developing information systems with UML.* Addison-Wesley, Harlow, 2001.

[9] Object Management Group, *OMG Unified Modeling Language Specification version 1.3.* Available at http://www.omg.org

[10] Robinson K. and Berrisford G. *Object-oriented SSADM.* Prentice Hall, 1994.

[11] Rumbaugh J. et al, *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, 1991.

[12] Rumbaugh J., Jacobson I., and Booch G., *The Unified Modeling Language Reference Manual.* Addison-Wesley, Reading, Mass, 1999.

[13] Wilkinson N.

[14] Wirfs-Brock R., Wilkerson B., Wiener L., *Designing Object-Oriented Software Englewood.* Prentice Hall, Englewood Cliffs London, 1990.

[15] Yourdon E., *Object-Oriented Systems Design: An Integrated Approach.* Prentice-Hall International, Englewood Cliffs, N.J, 1994.

Appendix IV:

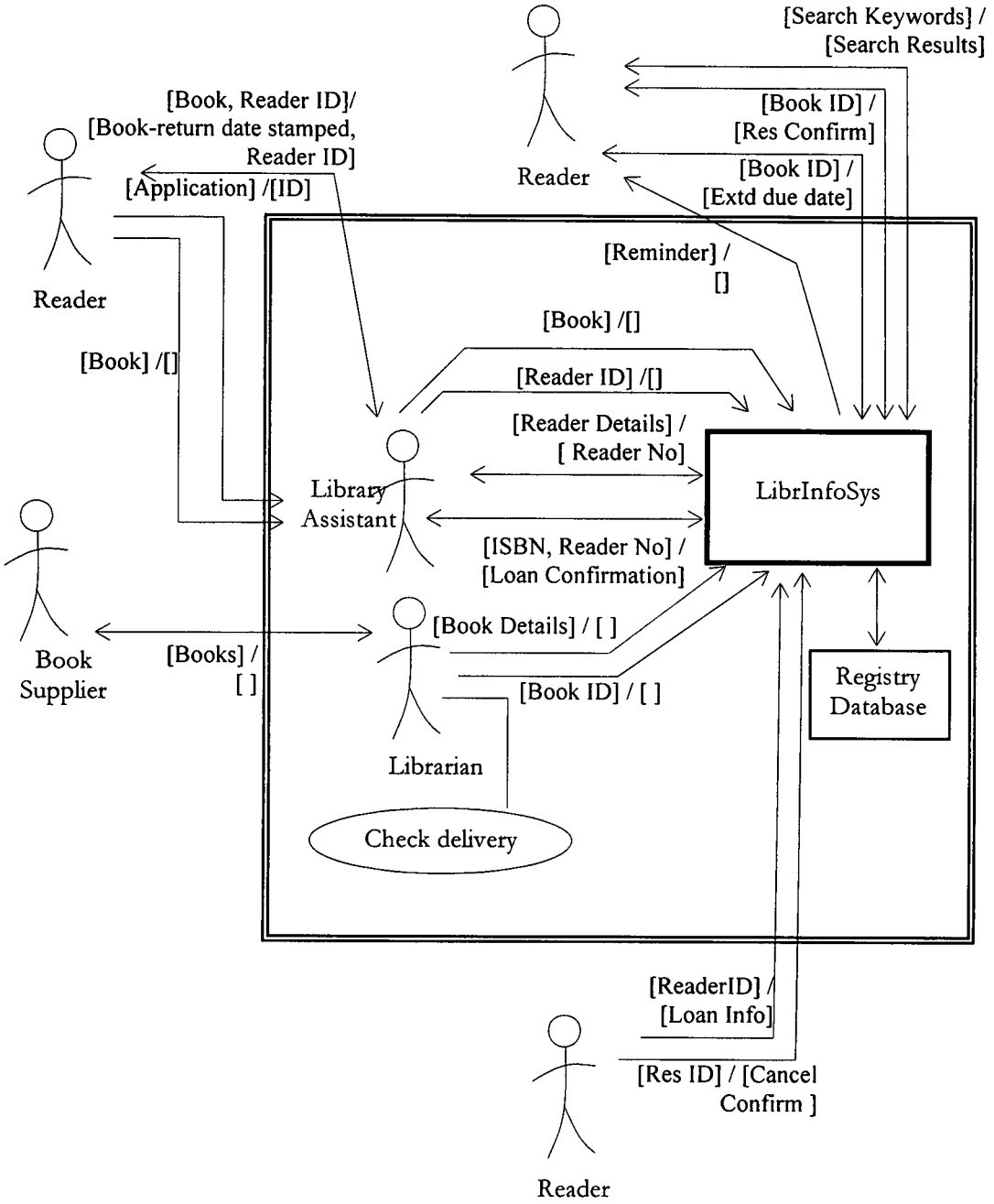
LIBRINFOSYS CASE STUDY

LibrInfoSys is a typical student library with the following key functionality:

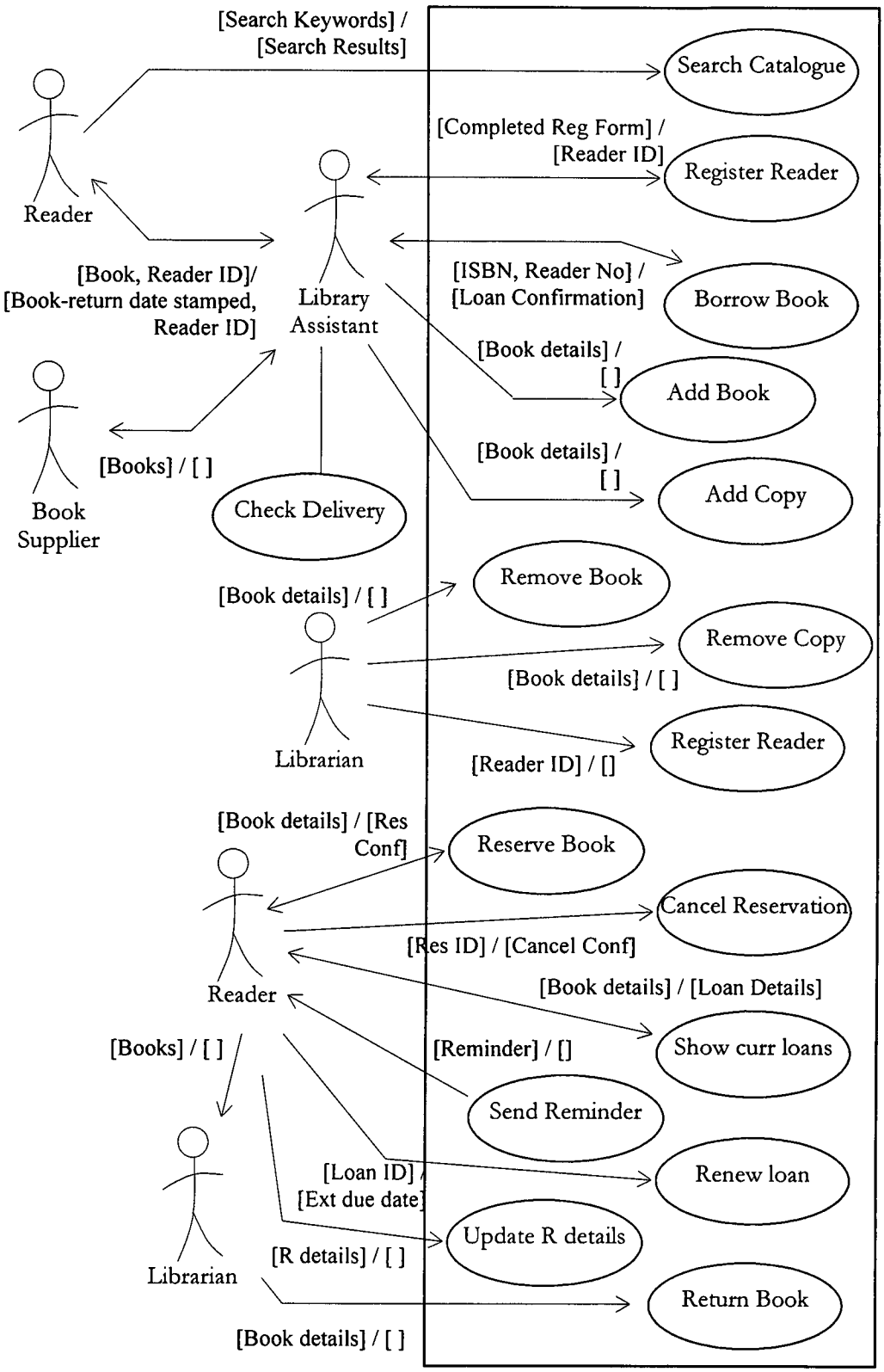
- Add Book
- Add Copy
- Register Reader
- Search Catalogue
- Borrow Book
- Return Book
- Renew Loan
- Reserve Book
- Cancel Reservation
- Show Reader's current loans
- Send Reminder
- Update Reader Details
- De-register Reader
- Remove Book
- Remove Copy

The following pages contain a selection of NAVITA diagrams developed for this application.

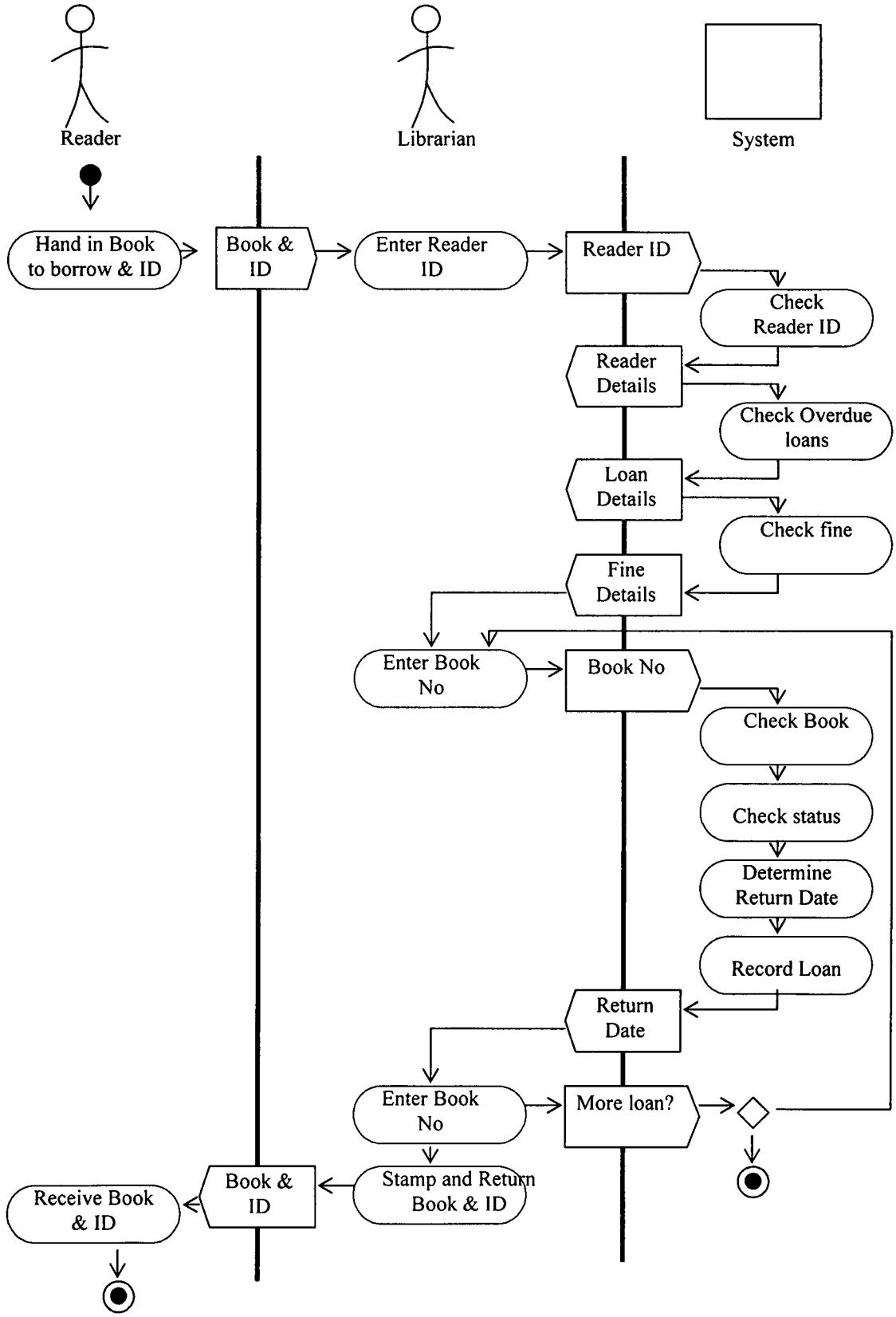
This is the NAVITA Context Diagram for LibrInfoSys.



The following is the MFD for LibrInfoSys,



This is the LFD for Borrow Book functionality unit.



The following is a simple LSL for Register Reader functionality.

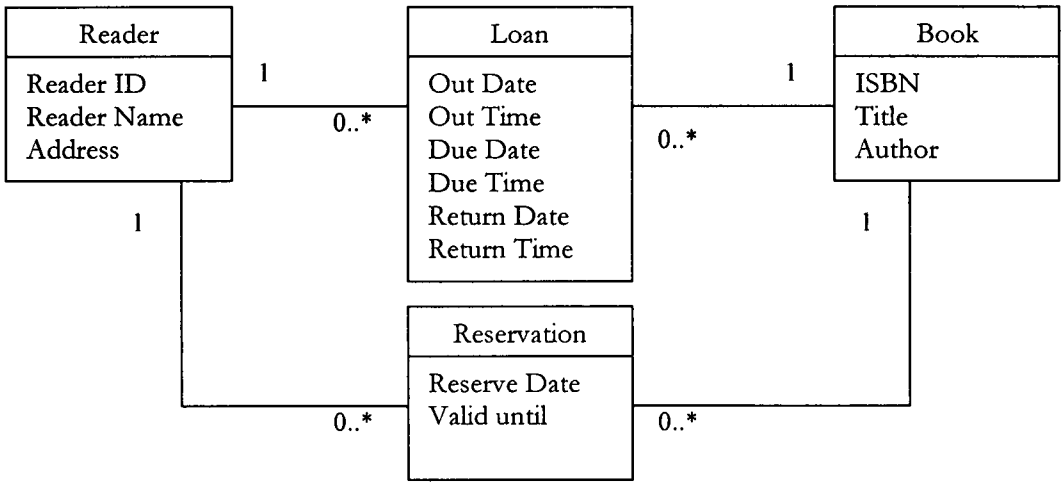
Register Reader

Personal Details:

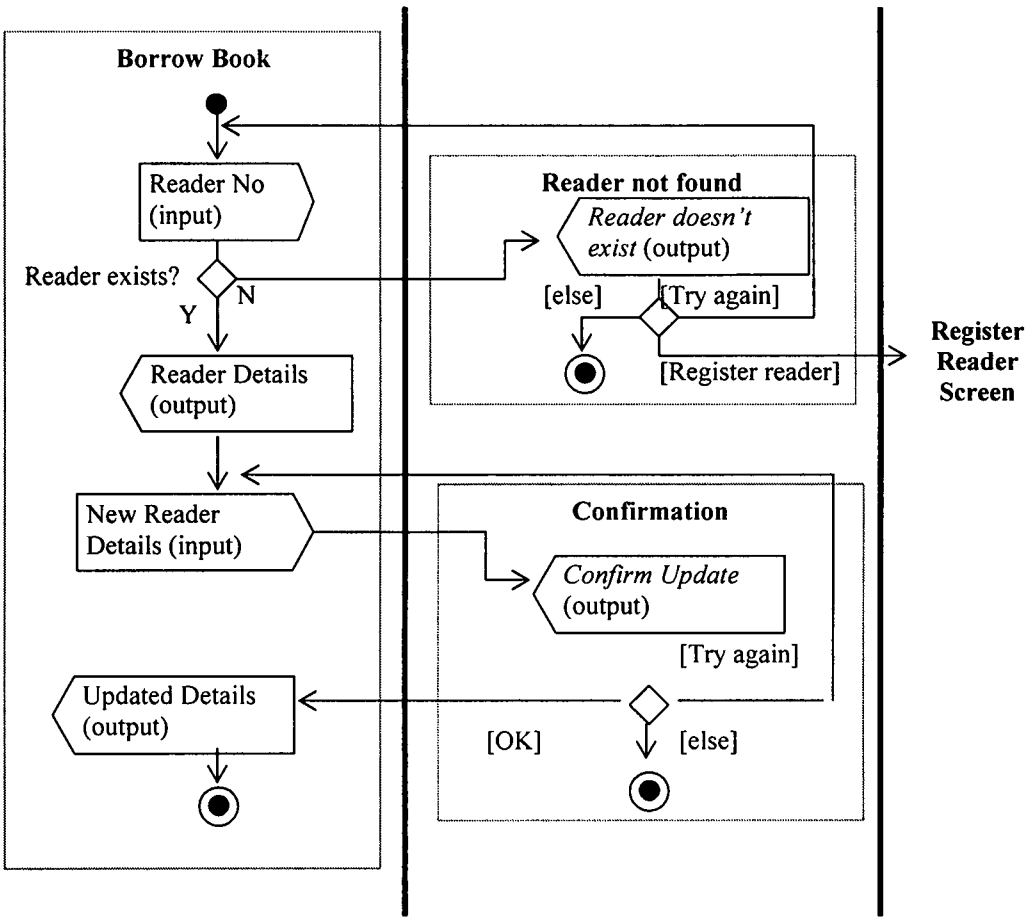
Reader name: ********
Address: **********
Other Details: ***

Reader ID: ----------

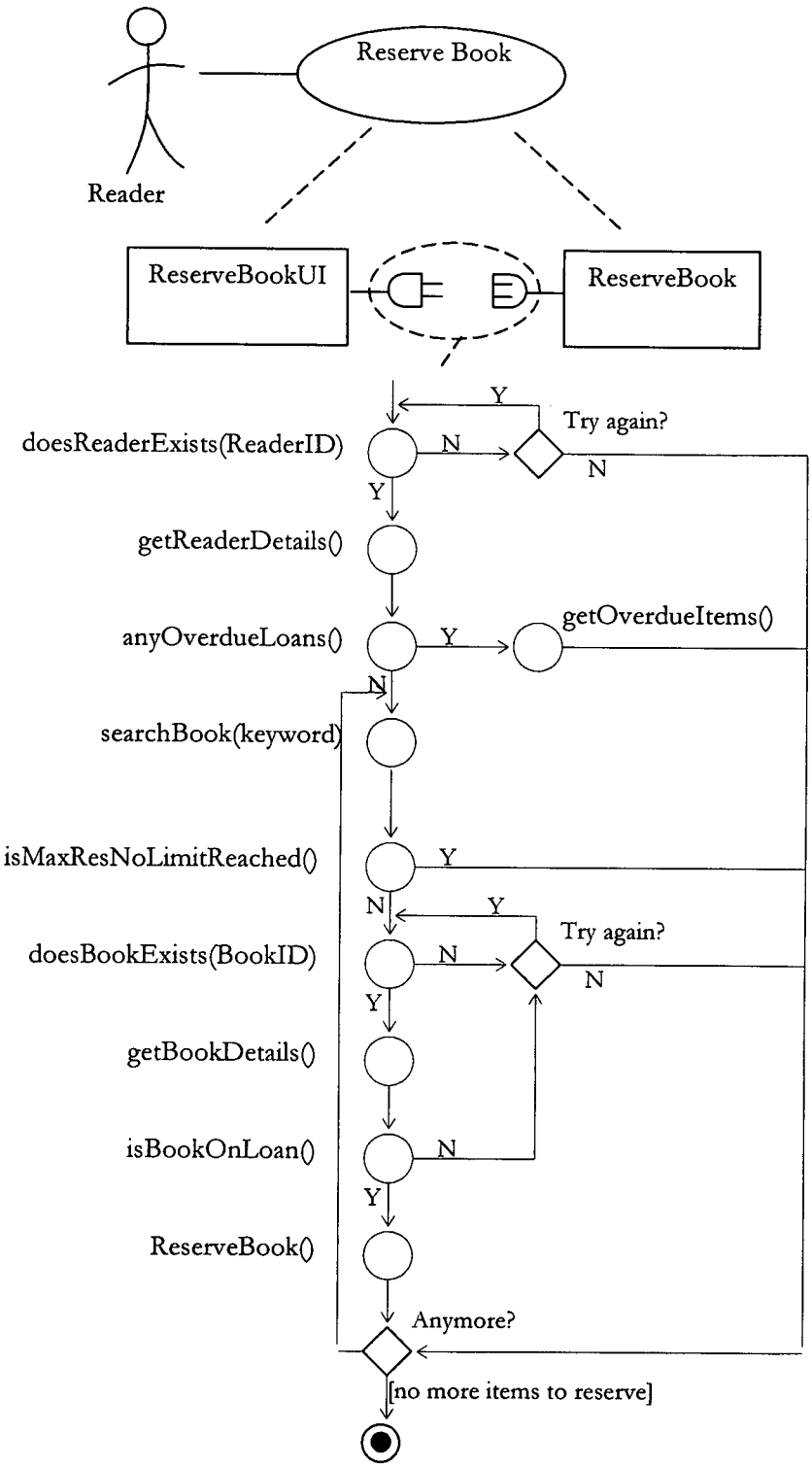
Main entities in the LibrInfoSys, together with their relationships are shown in the following diagram.



This diagram captures the dynamic interaction between the user and the system for Update Reader Details functionality unit.



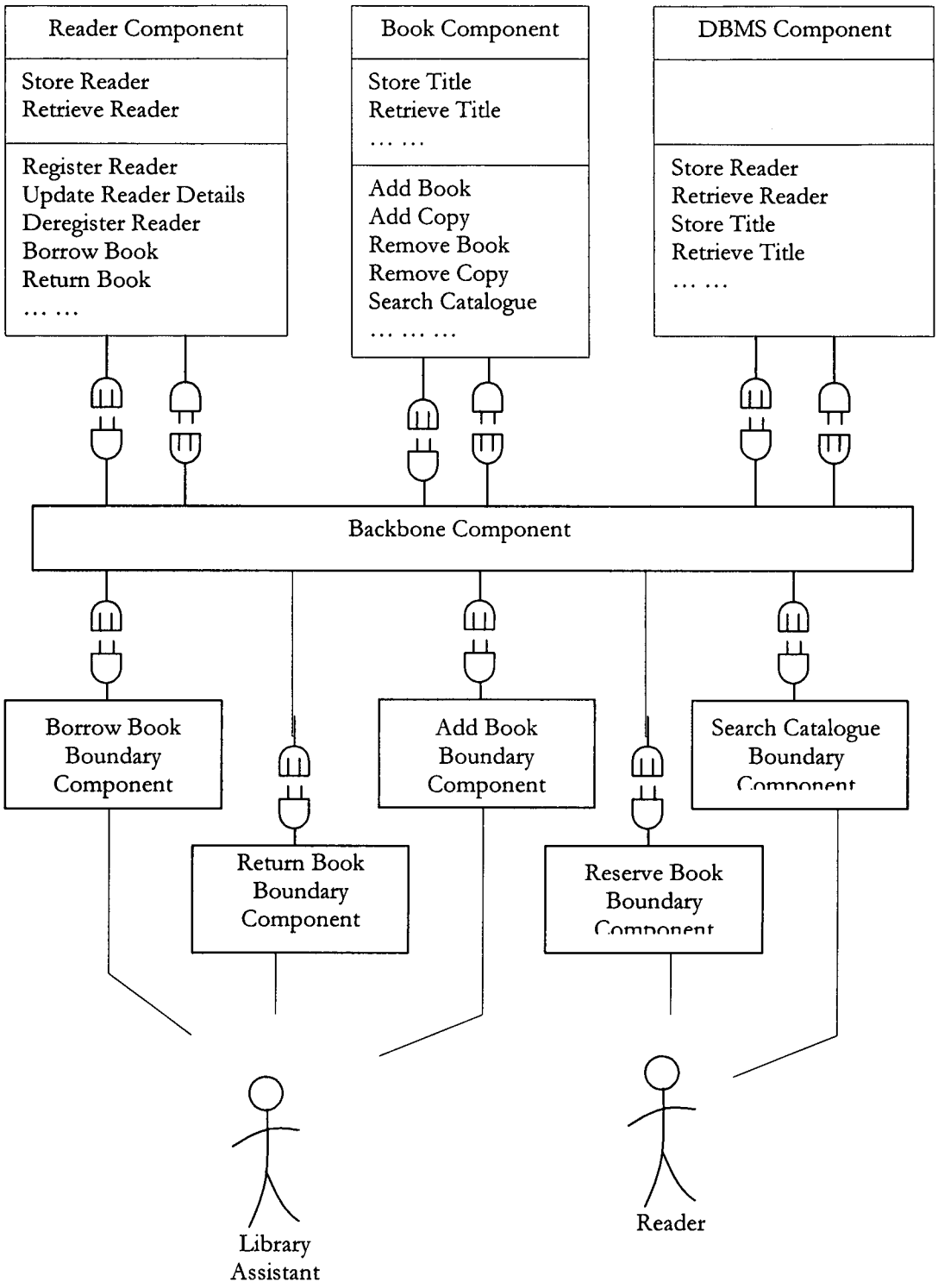
This is a protocol analysis for the communication between a boundary component and a business component for the Reserve Book functionality unit.



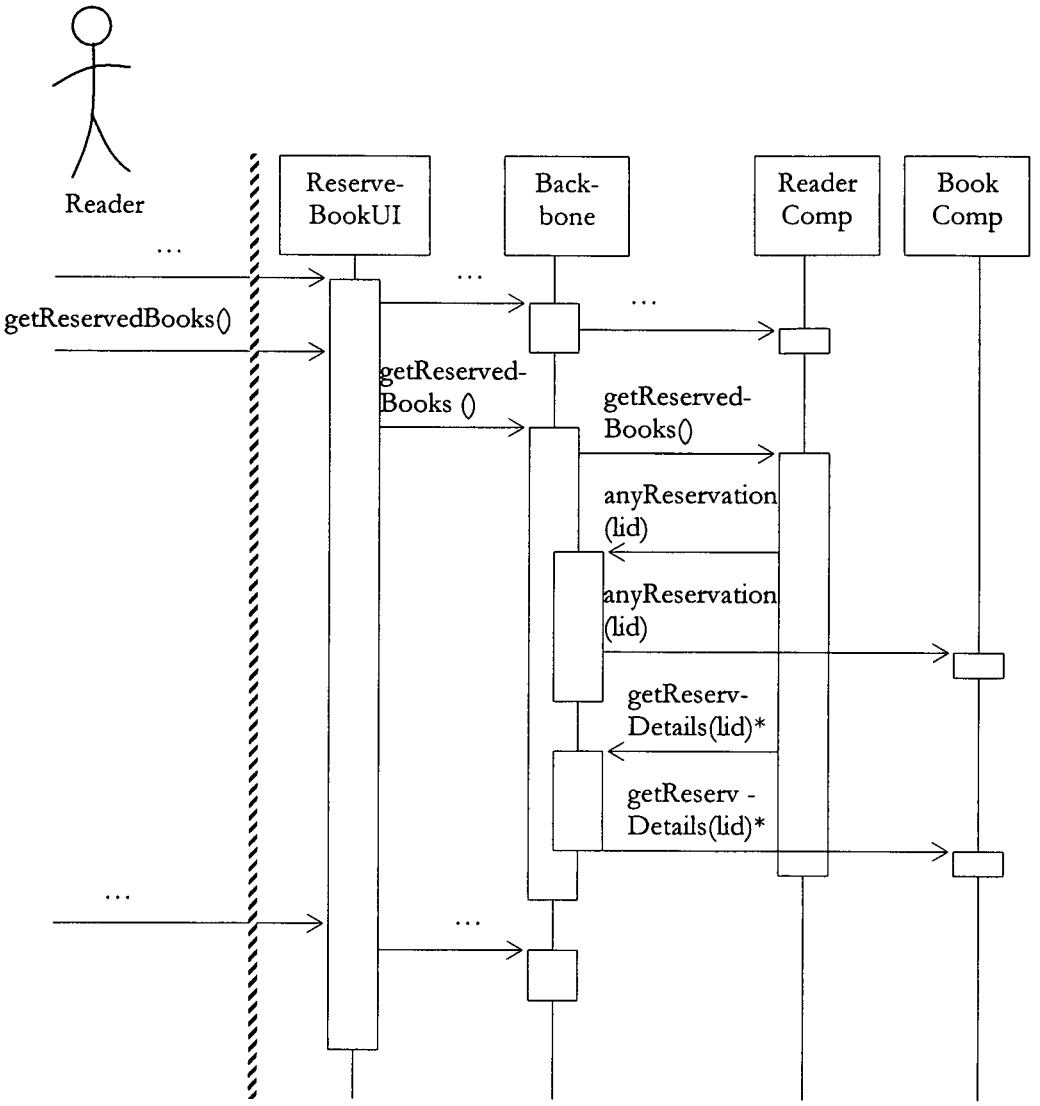
This FEM matrix captures the effects of functionality units on entity classes in LibrInfoSys.

	Reader	Reservation	Loan	Title	Copy	Author/Title	Author
Add Book				I L_1 L_3	I L_1	I L_2 L_3	I L_2
Add Copy				L	I L		
Register Reader	I						
Search Catalogue				R*	R*	R*	R*
Borrow Book	R L_1 M	[R M]	IL_1 L_2	R	R L_2	R	R
Return Book	R C_1		R M C_1 C_2	R	R C_2	R	R
Renew Loan	R		R M	R	R	R	R
Reserve Book	R L_1	IL_1 L_2		RL_2	R	R	R
Cancel Reservation	R C_1	C_1 C_2 D		RC_2	R	R	R
Show Reader's current loans	R		R*	R*	R*	R*	R*
Send Reminder	R		M	R	R	R	R
Update Reader Details	R M						
De-register Reader	R C_1 C_2 D	C_1	C_2				
Remove Book				R C_1 C_3 D	R C_1 D	R C_2 C_3 D	R C_2 D
Remove Copy				R C	R C D	R	R

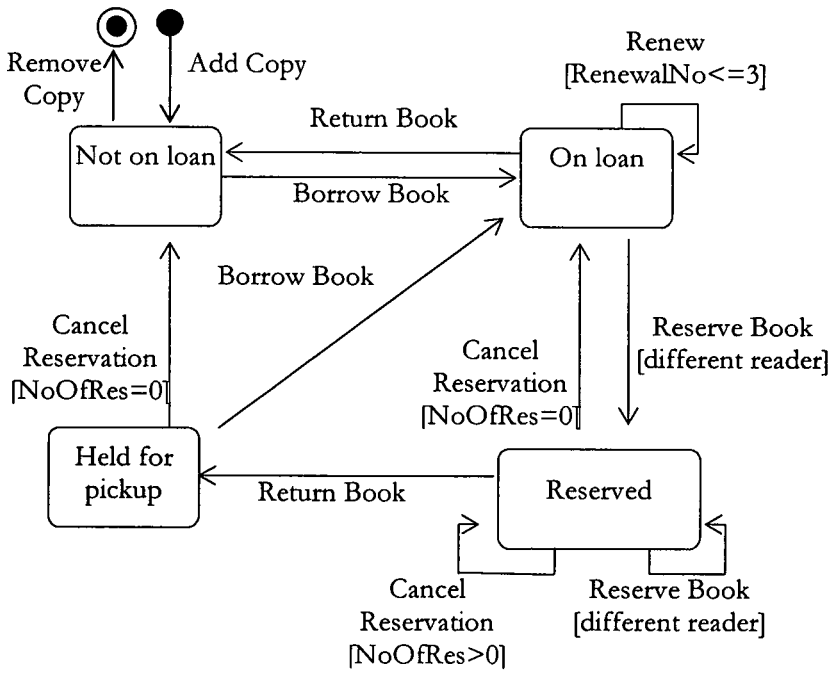
The following diagram describes the physical architecture of LibrInfoSys application.



This is a partial sequence diagram for Reserve Book showing how the backbone, Reader and Book components communication to realise the functionality unit.



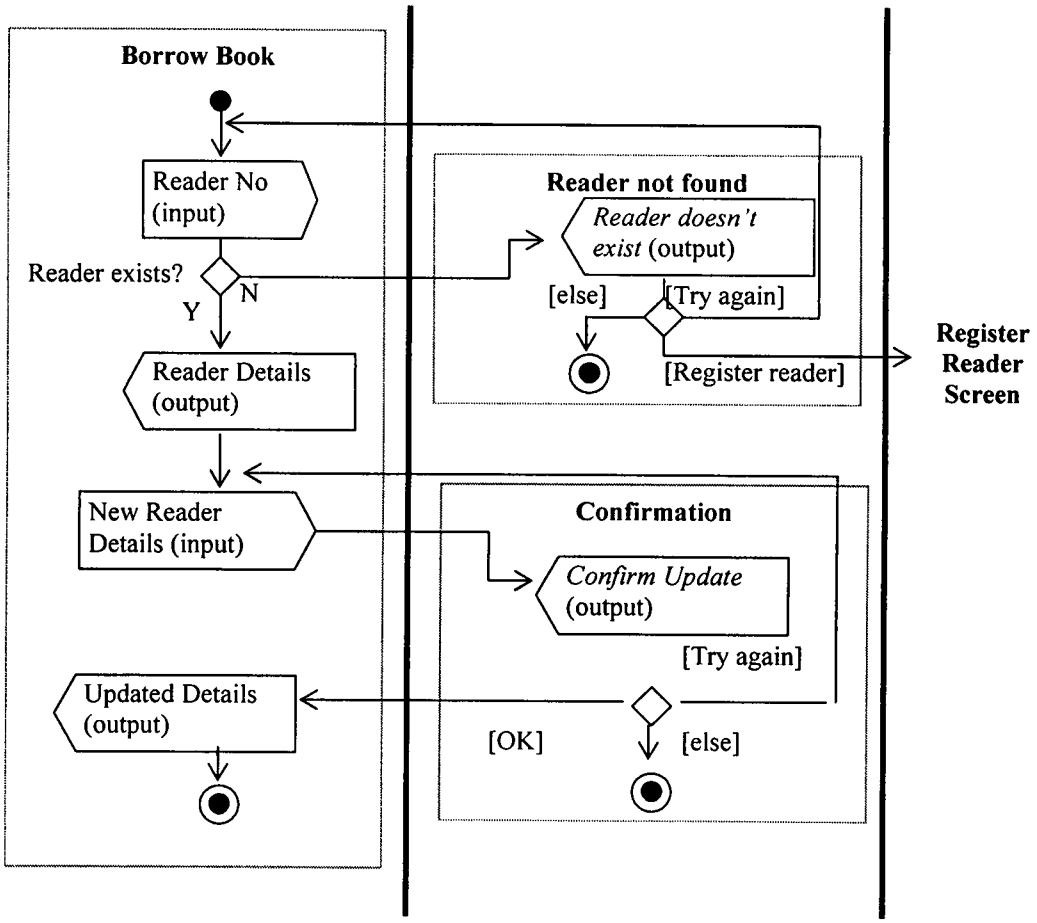
This STD shows the state changes in the Book component caused by various events.



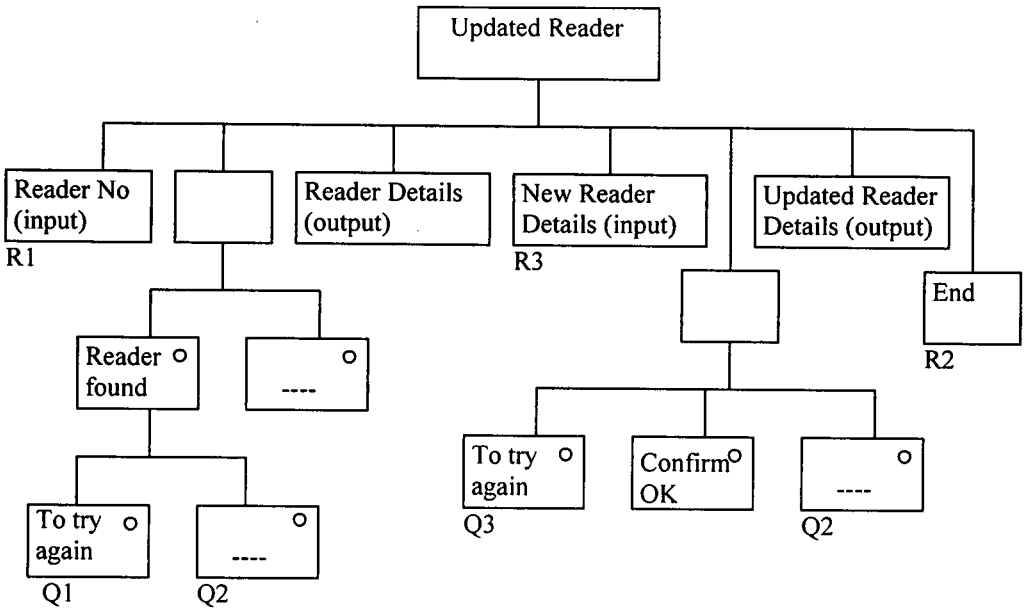
Appendix V:

1 A COMPARISON OF UML ACTIVITY-LIKE DIAGRAM AND STRUCTURED DIAGRAM

Two relatively simple diagrams can be used to illustrate that UML activity diagrams are less cluttered and easier to read than Structured diagrams. The following is a UML-activity like diagram used in NAVITA. The diagram is almost self-explanatory.



If the above diagram is converted into a Structured diagram, a diagram such as the following one will be necessary. With multiple quits and resumes, Structured diagram is relatively much more difficult to read than the previous UML Activity-like diagram. For this reason, the UML notation was preferred over the Structured one.



Appendix VI: Raw Data From The Repeatability Experiment

SSADM	Group 1	Group 2
Global Process Model	DFD	DFD
Global Information Model	LDM	ER Diagram
Global Interaction Model	Context Diagram	Context Diagram
Information-Process Contextual Models	EEM, ELH	ELH, EAM
Process-Information Contextual Models	None	IO Structur, Elementary process description
Process-Interaction Contextual Models	None	EAP, Enquiry function definition
Interaction-Process Contextual Models	None	ECD and EAM
Interaction-Information Contextual Models	ECD, User Role Matrix	None
Information-Interaction Contextual Models	None	None
Total Global Models Shown in the matrix	3	3
Acceptable Global Models in the matrix	3	3
Total Abstract Context Models	2	4
Acceptable Abstract Context Models	1	1
Total Detailed Contextual	2	4
Acceptable Detailed Contextual	1	1
UML		
Global Process Model	Use Case	Use Case
Global Information Model	Class Diagram	Class Diagram
Global Interaction Model	Activity	None
Information-Process Contextual Models	None	Activity and State Diagrams
Process-Information Contextual Models	Sequence and Collaboration Diagrams	Sequence and Collaboration Diagrams
Process-Interaction Contextual Models	None	None
Interaction-Process Contextual Models	None	None
Interaction-Information Contextual Models	State diagram, activity diagram	None
Information-Interaction Contextual Models	None	None
Total Global Models Shown in the matrix	3	2
Acceptable Global Models in the matrix	2 or 3	2
Total Abstract Context Models	2	2
Acceptable Abstract Context Models	0	0
Total Detailed Contextual	2	2
Acceptable Detailed Contextual	2	1 or 2
Commentary on comparison	Same coverage and both equally strong	SSADM has better coverage, UML has no interaction global diagram
No of students in the group	4	2
Overall Grade (out of 16)	6	6

SSADM	G3	G4
Global Process Model	DFD	DFD
Global Information Model	ER	LDM
Global Interaction Model	EAM	Context Diagram
Information-Process Contextual Models	EAM col, ECD	ELH, EAP
Process-Information Contextual Models	Entity, events in context diagram	ECD, EEM Column
Process-Interaction Contextual Models	UPM, entity	None
Interaction-Process Contextual Models	UPM, Entity	None
Interaction-Information Contextual Models	EAP, data store	None
Information-Interaction Contextual Models	None	None
Total Global Models Shown in the matrix	3	3
Acceptable Global Models in the matrix	2	3
Total Abstract Context Models	4	1
Acceptable Abstract Context Models	1	1
Total Detailed Contextual	4	3
Acceptable Detailed Contextual	1	2
UML		
Global Process Model	Use Case	Activity Diagram
Global Information Model	Class Diagram	Class Diagram
Global Interaction Model	None	State diagram, Use Case diagram
Information-Process Contextual Models	entities in use case realisation, collaboration diagrm	None
Process-Information Contextual Models	entities in collaboration/sequence diagrams, state diagram or activity diagram	None
Process-Interaction Contextual Models	None	Sequence and Collaboration diagrams
Interaction-Process Contextual Models	None	None
Interaction-Information Contextual Models	None	None
Information-Interaction Contextual Models	None	None
Total Global Models Shown in the matrix	2	4
Acceptable Global Models in the matrix	2	3
Total Abstract Context Models	1	2
Acceptable Abstract Context Models	1	0
Total Detailed Contextual	3	0
Acceptable Detailed Contextual	1 or 2	0
Commentary on comparison	SSADM has better coverage, but UML concepts are advanced	SSADM has good coverage and inter-model checking. UML Diagrams are fragmented
No of students in the group	3	3
Overall Grade (out of 16)	6	5

SSADM	G5	G6
Global Process Model	DFD	DFD
Global Information Model	ER	ERD
Global Interaction Model	Context Diagram	Context Diagram and Univesal Functional Model
Information-Process Contextual Models	ECD and EAP	EEM column, ELH
Process-Information Contextual Models	ECD and EAP	ECD and EAP
Process-Interaction Contextual Models	DFD and Context diagram	None
Interaction-Process Contextual Models	DFD and Context diagram	None
Interaction-Information Contextual Models	Context and ER diagrams	None
Information-Interaction Contextual Models	Context and ER diagrams	None
Total Global Models Shown in the matrix	3	4
Acceptable Global Models in the matrix	3	3
Total Abstract Context Models	6	1
Acceptable Abstract Context Models	2 or 3	1
Total Detailed Contextual	2	3
Acceptable Detailed Contextual	2	2
UML		
Global Process Model	Use Case	Use Case
Global Information Model	Class Diagram	Class Diagram
Global Interaction Model	Deployment diagram	Package Diagram
Information-Process Contextual Models	Diagram no name	Collaboration and Sequence Diagram
Process-Information Contextual Models	Sequence and Collaboration diagrams	None
Process-Interaction Contextual Models	Activity Diagram	None
Interaction-Process Contextual Models	None	None
Interaction-Information Contextual Models	None	None
Information-Interaction Contextual Models	None	None
Total Global Models Shown in the matrix	3	3
Acceptable Global Models in the matrix	2	3
Total Abstract Context Models	3	0
Acceptable Abstract Context Models	0	0
Total Detailed Contextual	2	2
Acceptable Detailed Contextual	1 or 2	0
Commentary on comparison	SSADM provides all three global models, context diagram is unique. UML has fewer contextual diagrams	Both have good number of global models, but UML has fewer crosschecks.
No of students in the group	4	4
Overall Grade (out of 16)	10	8

SSADM	G7	G8	G9
Global Process Model	DFD	DFD	DFD
Global Information Model	LDS	LDS	LDS
Global Interaction Model	Context Diagram	Context Diagram	Context Diagram
Information-Process Contextual Models	EEM column, and ELH	ELH?	ECD and EAP
Process-Information Contextual Models	None	ELH	EEM, LDS-Entity Cross References, ELH
Process-Interaction Contextual Models	None	I/O Structure	None
Interaction-Process Contextual Models	None	None	None
Interaction-Information Contextual Models	I/O Structure and ECD	None	User Role-Function Matrix, IO Structure
Information-Interaction Contextual Models	None	None	None
Total Global Models Shown in the matrix	3	3	3
Acceptable Global Models in the matrix	3	3	3
Total Abstract Context Models	2	3	3
Acceptable Abstract Context Models	0	1	1
Total Detailed Contextual	2	2	3
Acceptable Detailed Contextual	0	2	2 or 3
UML			
Global Process Model	Activity Diagram	Use Case and Activity Diagram	Use Case Diagram
Global Information Model	Class Diagram	Class Diagram	Class Diagram
Global Interaction Model	Use Case Diagram	None	None
Information-Process Contextual Models	None	Sequence and Collaboration diagrams	Activity and State Diagrams
Process-Information Contextual Models	Sequence and Activity Diagrams	State and activity diagram	Sequence and Collaboration Diagrams
Process-Interaction Contextual Models	None	None	None
Interaction-Process Contextual Models	None	None	None
Interaction-Information Contextual Models	State and Collaboration Diagram	None	None
Information-Interaction Contextual Models	None	None	None
Total Global Models Shown in the matrix	2	3	2
Acceptable Global Models in the matrix	1	2 or 3	2
Total Abstract Context Models	2	2	2
Acceptable Abstract Context Models	1	1	0
Total Detailed Contextual	2	2	2
Acceptable Detailed Contextual	1	1	1 or 2
Commentary on comparison	No attempt to show comparison, only generaly commentary	Similar in terms of coverage and consistency checking, but SSADM is marginally better	UML has fewer contextual diagrams
No of students in the group	3	3	3
Overall Grade (out of 16)	10	12	9