Domain Oriented Object Reuse Based on Generic Software Architectures

Adil Al-Yasiri

A thesis submitted in partial fulfilment of the requirements of Liverpool John Moores University for the degree of Doctorate of Philosophy

May 1997

THE FOLLOWING HAVE NOT BEEN COPIED ON INSTRUCTION FROM THE UNIVERSITY

Figure 2.1 page 21

Figure 2.2 page 30

Figure 2.3 page 31

Figure 2.4 page 32

Figure 2.5 page 34

Figure 2.7 page 39

Figure 2.8 page 42

Figure 2.9 page 43

Figure 2.10 page 48

Table 3.1 page 56

Figure 3.4 page 61

ORIGINAL COPY TIGHTLY BOUND

Abstract

In this thesis, a new systematic approach is introduced for developing software systems from domain-oriented components. The approach is called *Domain Oriented Object Reuse (DOOR)* which is based on domain analysis and *Generic Software Architectures*. The term 'Generic Software Architectures' is used to denote a new technique for building domain reference architectures using *architecture schemas*. The architecture schemas are used to model the components behaviour and dependency. Components dependencies describe components behaviour in terms of their inter-relationships within the same domain scope. DOOR uses the architecture schemas as a mechanism for specifying design conceptions within the modelled domain. Such conceptions provide design decisions and solutions to domain-specific problems which may be applied in the development of new systems.

Previous research in the area of domain analysis and component-oriented reuse has established the need for a systematic approach to component-oriented development which emphasises the presentation side of the solution in the technology. DOOR addresses the presentation issue by organising the domain knowledge into levels of abstractions known to DOOR as sub-domains. These levels are organised in a hierarchical taxonomy tree which contains, in addition to sub-domains, a collection of reusable assets associated with each level. The tree determines the scope of reuse for every domain asset and the boundaries for their application. Thus, DOOR also answers the questions of reuse scope and domain boundaries which have also been raised by the reuse community.

DOOR's reuse process combines *development for reuse* and *development with reuse* together. With this process, which is supported by a set of integrated tools, a number of guidelines have been introduced to assist in modelling the domain assets and assessing their reusability. The tools are also used for automatic assessment of the domain architecture and the design conceptions of its schemas. Furthermore, when a new system is synthesised, components are retrieved, with the assistance of the tools, according to the scope of reuse within which the system is developed. The retrieval procedure uses the components dependencies for tracing and retrieving the relevant components for the required abstraction.

Acknowledgement

I would like to express my gratitude to my supervisors Dr. Carl Bamford and Dr. Keith Whitely for their guidance and support throughout the course of this project. I also would like to thank Dr. M. Ramachandran for his supervision at early stages of the research. I would like to thank the school of Computing and Mathematical Sciences in Liverpool John Moores University for providing funding for this research.

Special acknowledgement goes to my parents and family to whom I am indebted and without their support and encouragement this thesis would not have been completed.

TABLE OF CONTENTS

List of Acronyms

3

3-D
Three Dimensional Model 19

A

ADT
Abstract Data Type 23
AI
Artificial Intelligence 68
AIS
Adaptive Intelligent Systems 74
ARPA
Advanced Research Projects Agency 71
ASM
Abstract State Machine 24

D

DA
Domain Analysis 35
DICAM
Distributed Intelligent Control and Management 71
DOOR
Domain-Oriented Object Reuse 81
DSSA
Domain Specific Software Architectures 71

E

EDLC
Evolutionary Domain Life Cycle 47

H

HP
Hewlett-Packard 19
HTML
Hyper Text Markup Language 172

M

MVC
Model-View-Controller 54

N

NASA
National Astronomy State Agency 33
NIH
Not-Invented Here 15

O

O-O
Object-Oriented 33
OOA
Object Oriented Analysis 61
OOD
Object Oriented Design 61
OOPSLA
Object Oriented Programming Systems, Languages and Applications 62

R

REBOOT

Reuse Based on Object-Oriented Techniques 18

S

SAAM

Software Architecture Analysis Method 78

SEL

Software Engineering Laboratory 33

SSADM

Structured System Analysis and Design Method 30

W

WWW

World Wide Web 173

List of Tables

List of Illustrations

Chapter One

1. Introduction.

1.1 The Software Crisis.

The software industry has come a long way in a short period of time and the process of writing software has changed from an art to an engineering discipline. Programming and design methods have changed a great deal and, in consequence, have changed the way in which modern software is written. New software engineering approaches have emerged in order to make the software development process systematic. With the increasing demand on software systems, (because of the rapid development in hardware) they have become much more complicated and forced their way into all branches of modern life. Payroll systems, air traffic control systems, word processors, numerical controlled machines and many more examples, are systems run by software. The size of these systems increasingly becomes larger and larger. Nowadays, we can find very large systems with millions of lines of code.

The cost of software development forms the major part of the overall cost of any information system project. This is because of the nature of software development, since it is a labour-intensive process which restricts productivity and increases cost. Furthermore, maintaining software systems is a difficult process and sometimes painful. For these reasons among others, the need for reliable, well-engineered software has become vital to reduce cost and increase productivity and maintainability.

The term "software crisis" refers to the problems that are encountered in the development of computer software [Booch 1986]. The problems are not limited to software that does not function properly but in the way software systems are developed and maintained. The questions are: are we really in a crisis? And what are the problems that characterise the crisis? Most of the problems suffered by software systems that characterise the crisis in the software industry are problems that deal with managing the development process. The main problems are the management of cost, quality and productivity within the software projects. Software systems are built from scratch every time new ones are developed. This yields to the fact that there is no effective way to accurately estimate the time and cost of developing new systems. Quality and productivity cannot effectively be measured and controlled for the same reason. Hence cost and implementation time normally run over the estimated plan causing an increasing pressure to meet the project deadline and hindering the quality of the product. When it comes to maintenance, the problems are even more severe. System maintainability was not a factor in the design of the existing systems, therefore maintaining these systems can be very difficult. A high proportion of the software cost is consumed in the maintenance stage.

Problems associated with the software crisis have been caused by the character of software itself [Pressman 1992]. Software is a logical rather than a physical system element and its realisation is usually seen as a challenge to the people who develop it. The intellectual challenge of software development is one cause to the crisis, however the way in which software has been developed and the people who are in charge of doing it bear a large share of the responsibility.

New techniques for the software development process have been introduced in order to reduce cost and increase productivity. Structural methods, object oriented techniques, rapid prototyping, and software

reuse are examples of these techniques. Among these, software reuse is a newly adopted technique which seems promising. In the next section, software reuse is introduced and its advantages are underlined.

1.2 Software Reuse, Why?

Reuse means re-applying knowledge gained through the development of one system to another system in order to reduce the effort of development and maintenance of the other system [Biggerstaff and Perlis 1989]. Reuse could include the reuse of design methods and decisions and code reuse. Code reuse is not a new concept as the use of functions, subroutines and libraries in FORTRAN programming forms some kind of reusability. Other examples of reuse are found in the UNIX system through the use of filters and pipelines that enable the user to connect and interface several commands together to form new functionality. Lex and Yacc are real examples of code reuse as they can be used to generate compilers from a definition language.

Adopting a reuse approach to software development means making maximum use of existing software components whenever that is possible. As expected, the major advantage of this approach is reduction in the overall development cost. Fewer software components need to be specified, designed and implemented in the development process of the new system. However, the exact amount of reduction in cost is difficult to calculate. The estimated unnecessarily-developed code in data processing applications is about 60% of the overall code, and could be standardised and reused [Hall and Boldyreff 1991]. Expectedly, reuse will sharply increase productivity and the effort spent on developing reusable software is worth taking. For example, the Japanese reported in 1980 that they had achieved an overall increase in productivity of 14% per year over several years by introducing reuse in the development process. They view the process as a manufacturing process, hence the term "Japanese software factory".

Other advantages can be gained from adopting systematic reuse in the development process, which are [Sommerville 1992]:

1- Increasing the system reliability: Reused components, which have been applied in working systems are more reliable than new components as new components have not been tested in actual working environments.

2- Reducing overall risk: The uncertainty of estimating the cost of developing new components is eliminated by reusing existing components. This forms better grounds for project management in reducing the risk in cost estimation.

3- Making effective use of specialists: In the process of developing software systems, application specialists are referred to as sources of information. They join the project for a short time and they often do the same job. These specialists can develop reusable components which encapsulate their knowledge.

4- Developing standardisation with reusable components: Developing reusable components forms a good opportunity for including some standard objects which can become familiar for all users. For example, applying reusable components providing menus to different applications means that all applications present the same menu format to users.

5- Reducing development time: Reusing components speeds up system production because both development and validation should be reduced. This is very important in marketing software systems.

1.3 What is a Domain and What is Domain Analysis?

Developing software systems from reusable modules requires that software components must first be built in a form suitable for reuse. This fact leads to the division of the development process into two equally important stages; *software development for reuse* and *software development with reuse.*

In software development for reuse, every component constructed for future reuse must be designed such that it could be used over the application to which that component belongs. We refer to a class of similar systems as a *domain*. The underlying feature that define domains is the similarity of the systems that belong to it. These could be in the form of one or more of these features: functions, objects, sub-systems or design structures. Domains need to be analysed and modelled in order to identify the similarity that characterise the reusable objects in them. The necessary information, that is required in the analysis process, can be obtained from different sources like technical literature, existing systems, customer surveys, human expertise and current and future requirements. The process of knowledge acquisition, identifying and analysing reusable objects of a class of similar systems is called *domain analysis*. With domain analysis, we try to model the whole domain rather than the system under development only. Modelling the domain means searching for any common objects and features in the application and providing a specification framework for components with potential reusability.

The first introduction to domain analysis was made by Neighbors when he described an approach to software reuse which is known as the *Draco* approach [Neighbors 1984]. He referred to the term domain analysis as "the activity of identifying the objects and operations of a class of similar systems in a particular problem domain". The Draco approach tackled the problem of domain analysis at the organisation level. The development life-cycle was extended to include a phase for future reuse. New roles for *domain analysts* were introduced to carry out domain analysis tasks in the development process.

1.4 Objectives and Contributions of the Work

There are still some problems in the way software components are referenced and retrieved. Components' descriptions must include some

information about their functionality and scope before they could be used. On the other hand, components cannot be specified or reused in isolation of other components in the domain. The way components are normally designed and implemented is influenced by the scope of application and their interaction with other entities in the domain. Component indexing and retrieval mechanisms ignore this critical issue and treat components as stand alone entities that could be located and retrieved for any application. This view is based on generic reusable components.

Another problem that faces the reuse community is how domains could be analysed and structured for the reuser to make effective use of the domain resources. Current domain analysis methods are mainly ad-hoc in which reuse is opportunistic rather than systematic (see chapter two for discussion). When domains are analysed, there is no clear idea about the outputs of the process or how to achieve the goal. In some cases, domain architectures are used for introducing the problem domain. The architectures do not provide the solution to the problem. Furthermore, since there is no systematic approach to domain analysis, there is a lack in the tool support for domain analysis and domain modelling.

In this thesis, a domain-oriented approach to software reuse is proposed for solving the above problems. It is based on the concept of application scope. When domains are analysed, they are divided into a hierarchy of abstraction levels. These levels represent the scope for reuse of the reusable components in the domain. The approach allows components to be identified and designed for reuse within a specific scope. When they are retrieved, the scope is used as a guide for locating and tracing components from a domain knowledge base. The approach supports reuse through the reuse of design conceptions within the domain. Thus it is a solution-based approach in which the solutions to the domain problems are encapsulated in the reusable components.

The approach is based on the following elements:

1. A *domain infrastructure*, that supports the development of new systems. The infrastructure comprises a domain taxonomy, reusable components and a reference architecture.
2. A *technology* which helps in developing and refining the infrastructure. This is encapsulated in a number of *Generic Architectural Models* which are used for describing the reference architecture.
3. A *process* and *guidelines* for building the infrastructure (using the technology) and for synthesising new systems. The process is divided into *Domain Engineering* and *Application Engineering* phases.
4. A supporting tool for automating and validating the infrastructure development process.

Components are modelled in terms of their scope, behaviour and their interaction with other components in the domain. The relationships are modelled using the technology of generic software architectures. These architectures are built using pre-described architectural models that are used for modelling the design conceptions in the domain. Architectures are also used for tracing components in the domain knowledge base by tracing the relationships among reusable components.

The process of building software systems in this approach is designed in such a way to make it as systematic as possible. This is done by providing a list of design guidelines for building the domain model and checking the overall design. A number of guidelines are also set for assessing the reusability of the domain components as well as the architectures.

The novelty of this approach could be summarised in the following points:

1. The approach provides a new way for classifying and structuring the domain abstractions which is used for defining the component scope of application. Thus it provides a representation method of the domain knowledge as well as an indexing scheme for its components.

2. Components are described using a new model called the 3-D model (scope, behaviour and reaction space) which helps in describing the component's functionality and constraints.
3. A technology for modelling the domain architectures using generic architectural models. The technology is named *Generic Software Architectures* which is used for modelling the components' relationships and dependency within the domain. These relationships are also used for tracing components in the domain knowledge base.
4. A set of design guidelines for classifying the domain and building the domain architectures. These guidelines make the approach more systematic and help with assessing the components' reusability as well as the architecture validity. The guidelines are developed through experience gained from applying the approach to a number of exemplar domains. Further work may be needed for developing more guidelines as the approach is extended in the future.
5. An integrated tool for supporting classifying and modelling domains, building domain architectures, tracing and retrieving components and reuse assessment.

Some of the work developed in this thesis has been presented in a number of publications [Al-Yasiri and Ramachandran 1994, Ramachandran and Al-Yasiri 1994A, Ramachandran and Al-Yasiri 1994B].

1.5 Outline of the Thesis

The thesis is organised as follows. Chapter two introduces the main issues of software reuse and domain analysis within the software development process. It also reviews current research in software reuse, components retrieval and domain analysis methods. Chapter three discusses software architectures and object-oriented design patterns (sometimes called micro architectures) and their support for reuse. Current research in these two

areas is also reviewed. Chapter four presents an overview of the proposed approach with discussion about the domain classification and components modelling. Chapter five explains the technology of the generic software architectures, relationships between components and constraints. In chapter six, the process of modelling domain and building domain reference architectures is described and a number of design guidelines are introduced for executing the process. Chapter seven describes the tool support and presents a case study for applying the approach to a real-world domain. The thesis is concluded in chapter eight where the approach is critically assessed and future extension to the work is suggested.

Chapter Two

2. Software Reuse and Domain Analysis.

2.1 Introduction

In this chapter, the concepts of software reuse and domain analysis are introduced. Firstly, the factors that should be considered in software development from reusable modules are introduced. More specifically, composition based reuse and component retrieval mechanisms are reviewed in section 2.3. Later in the chapter, a review of domain analysis and domain analysis methods will follow with a critical comparison between different methods.

2.2 Reusability Issues and Software Development

Software reuse has been widely publicised and researched over the past few years because of its obvious benefits (see chapter one). However, there is little available evidence to suggest that systematic reuse is practised on a wide scale. This is because of a number of obstacles (both technical and managerial) that inhibit reuse. Some of these obstacles are connected to the way we write software components and others are connected to how we use them.

Gautier and Wallis define reusability as :

> "... a measure of the ease with which a software component may be used in a variety of application contexts" [Gautier and Wallis 1990]

Reusability is determined by the way components are written and how they are used. This leads to looking at the problems from two points of view; software development for reuse and software development with reuse.

2.2.1 Software Development for Reuse.

Adopting a component-based approach to reuse requires a library of software components already existing. Previous research showed that 40-60% of actual program code was repeated [Horowitz and Munson 1984, Lanergan and Grasso 1984]. Such observations among the software community have led to a common misconception, that components are available in existing systems [Sommerville 1992]. In fact, components must first be designed for reuse before they can be reused. This means they have to be generalised to satisfy a wider range of requirements.

When a software component is developed for reuse, there are a number of factors that affect its reusability; some are technical and some are managerial factors. On the managerial side, developing generalised components is more expensive than developing components for a specific purpose so increases project costs. As the principal role of project managers is to minimise costs, they are understandably reluctant to invest extra effort in developing components which will bring them no immediate return. The process requires an organisational policy decision to increase short-term costs for long-term gain. The organisation rather than individual project managers must make such decisions. The difficulty lies in how well senior managers can see the long-term benefits of such investments.

Technically, components must be designed to serve a well specified abstraction in the application domain. Sommerville lays two rules to assess the reusability of a component [Sommerville 1992]:

1. How well does the component represent an application domain abstraction?
2. Has the component been written so that it is generalised and adaptable?

These rules are useful if assuming that the developer and the reuser of the component are both experts in the application domain. The first rule is relevant but we cannot measure the precision of the component representation without understanding the domain first. The second combines adaptability and generality which are difficult to achieve at the same time. It could be more useful if the component exhibits completeness and flexibility. By completeness we mean that a component encapsulates all relevant features in a specific domain abstraction and nothing more. Flexibility means a component has the ability to evolve as the domain evolves.

In our view, more fundamental questions should be asked to assess the reusability of a component.

1. How accurate can we describe the domain abstraction?
2. How well does the component represent the solutions to that abstraction?
3. Can we comprehend the component behaviour such that it is easy to use and modify?

The answer to the first question requires a comprehensive modelling and presentation of the domain abstraction. Questions two and three deal with the modelling of the component behaviour and in order to answer them we need to model them with reference to their scope for reuse. Chapters four and five will explain our approach to modelling application domains and components' behaviour.

In the light of this discussion, some guidelines are needed for writing software components such that they have better reusability. In principle,

reusable components are designed around the principles of abstraction and information hiding. Some work has already been done to define some guidelines for component writing. Most of these guidelines are language dependent, especially targeting the Ada language. One work presented in [Gautier and Wallis 1990] set a number of Ada reusability guidelines and classified them into the following categories: design guidelines, generic components, exceptions and tasks. Obviously these guidelines are relevant to Ada as they deal with features supported by Ada. However some of the design guidelines could be generalised for other languages which are similar to Ada (strongly-typed languages) such as C++ or Object Pascal. As an example, consider the following guideline presented in [Gautier and Wallis 1990]:

> *Avoid specifying a package in such a way that all implementations of that package will have to maintain internal state.*

This guideline could be modified for defining C++ classes as follows:

> *Avoid specifying a class in such a way that all its member functions will have to maintain internal state (in other words, avoid declaring variables as static in such functions).*

Matsumoto suggests some general guidelines to make software modules reusable [Matsomoto 1984]. His list comprises the following characteristics:

1) Generality
2) Definiteness
3) Transferability
4) Retrieveability

The first two characteristics call for us to build components that are focused on a single abstraction. The next two characteristics are mainly issues of portability and library management.

Another work by [Booch 1987] sets a number of general guidelines for writing reusable components and some specific guidelines for writing Ada components for the domain of abstract data structures. Booch has developed an extensive classification structure for such components and discusses how generalised components can be implemented by applying his own guidelines. The general guidelines (set by him) state that software components should exhibit the best characteristics of any good piece of software which are:-

- maintainable
- efficient
- reliable
- understandable

Furthermore, Booch suggests three more desirable characteristics to be added to Matsumoto's list which are:

- Sufficient
- Complete
- Primitive

Booch's guidelines (in addition to Matsumoto's) deal with the outside view of a component and how components are utilised. However, the last three characteristics emphasise the generality issue of the designed component which is the major design consideration in Booch's components. We will come back to Booch's work in the next section when we discuss component-oriented reuse. Sufficiency means the component captures enough characteristics of the abstraction to permit meaningful interaction with it. Whereas completeness means the component's interface captures all characteristics of the component. On the other hand, a component specification must include primitive operations which can be efficiently implemented only with access to the underlying representation of the component; thus complying with the principles of abstraction and information hiding.

Further work has been done by Smith [Smith 1990] to set ten informal guidelines for enhancing reusability of software modules in the C and C++ languages. These are casual guidelines which the author does not claim to be applicable in all situations. In addition, they lack the depth under which Ada guidelines have been treated. Nonetheless, most of Ada reusability guidelines can be applied for C++ after modification, as we have outlined earlier in this section. In another work [Johnson and Foote 1988], some rules were suggested for designing reusable classes as a basis for reusable object oriented software. These are high-level design heuristics for designing reusable modules with no detailed information of how to implement the rules as the case with the Ada guidelines. As an example, consider the following rule for finding frameworks of interconnected classes *"Split large classes"*. This rule and its elucidation failed to show possible ways of splitting the classes; for instance, should they be broken into a number of sub-classes of another abstract class or should they be designed as parts of one composite class. Based on these rules, some object-oriented design guidelines were developed by [McGregor and Sykes 1992]. In their work, the guidelines are more focused and more specific. Some of them are a union of two or more rules presented by Johnson and Foote. Furthermore, the work shows, through examples, how to apply these guidelines as design choices (alternatives) to enhance reusability.

2.2.2 Software Development with Reuse

Developing new software systems from existing building blocks is called development with reuse. The main factor that motivates reuse within an organisation is the increase in productivity and competitiveness. However, several factors are still there inhibiting reuse. [Biggerstaff and Richter, 1987] outline some of these factors as:

1. Inadequate representation technology.
2. Lack of clear and obvious direction.
3. High initial capitalisation.

4. The Not-Invented-Here (NIH) factor.

In their opinion, the main factor that prevents the successful reuse of design information is the representation factor. They explained how the representation problem would prohibit the reusability of components. In this respect, the following features were identified to be needed in a representation approach or style:

- The ability to present knowledge about implementation structures in factored form.
- The ability to create partial specifications of design information that can be incrementally extended.
- The ability to allow flexible coupling between instances of designs and the various interpretations they can have.
- The ability to express controlled degrees of abstraction and precision (i.e., degrees of ambiguity)

The second factor concerns the debate between management and technologists. Managers usually are reluctant to invest in a new technology until they are certain of the best path. Technologist often take the initiative and explore the avenues researching for the best path. In this case, the nature of the problem is different; reuse is a multi-organisation problem and requires a library of software components before the pay-offs can be realised. Therefore, commitment on the management side is needed for building such libraries in order to anticipate the benefits of the technology. This also leads to the third factor which is high initial capitalisation where we need to invest a great deal of intellectual capital, real capital and time before we can benefit from the technology. The problem is not just the amount of investment needed for this commitment, but the time period required for the reorganisation of the company to accommodate reuse and reap the benefits of it. Normally, a reuse program takes between five to ten years to mature within an organisation [Lim, 1994]. This makes the problem harder on the management side, and

requires high gains in terms of productivity and competitiveness to justify such high profile commitment.

The NIH factor is a cultural problem that exists among software developers. Developers think that reusing existing components is limiting their creativity, in addition to the reduced confidence in the components that are developed elsewhere. Biggerstaff and Richter think that this problem is easily curable compared to some of the technical problems. Whereas they could be right in that it is less significant than the others, this work has shown that it is not easily curable because the cultural problems are linked to the technical and organisational issues. It will be easily curable if proper solutions to the other problems are found. They claim that the cure is up to the management to establish a proper reuse culture within the organisation, and when the developers practise reuse they will soon realise its benefits and find new challenges for proving their creativity. This is fine, but we already know that managers are reluctant to adopt the reuse approach. Moreover, we agree with their suggestion of rewarding successful reuse among individuals or within projects as an incentive to overcome this problem. Furthermore we believe that the problem needs a systematic approach to development with reuse where reuse is a planned process rather than ad-hoc and is supported by a number of clear steps for developing systems from existing components and a number of guidelines or heuristics to assess the validity of the design. In chapter five, we will present our solution to some of these problems based on the previous principles.

There are a number of examples of organisational approaches to development with reuse. The most successful approach was the Japanese software factories. In these factories, they integrated known techniques from different disciplines like source management, production engineering, quality control, software engineering and industrial psychology [Tajima and Matsubara 1984, Matsumoto, et al. 1980].

Matsumoto reported that they had achieved an increase in productivity of 14% per year over a number of years. The Toshiba factory, for example, [Matsumoto 1984] is using a set of well-known software engineering representation and design disciplines, and they enforce these design, environment and tool standards. The success of the software factories is attributed to the following reasons:-

- They have established a critical mass in the number of reusable components and programs (>1000) available to use and develop them.
- They have taken the separate phases in the software development process and assigned them to different departments within the software factory.
- They have developed an integrated set of tools and rigid standards to support reuse in the software production life-cycle. Because of the large number of users of the tools, their initial development cost can be economically justified.
- Their management is committed to this approach.
- Software reuse is part of their training process.

A British Aerospace project for large-scale reuse has been reported in [Hutchinson and Hindley 1988]. The project is aimed at supporting development of large, real-time, embedded systems. The approach is focused on a specific domain within which reusable software components are designed by isolating reuse attributes discovered during domain analysis. Reuse, in this approach, is identified in different levels which are based on the principle that every software entity is potentially reusable; whole system, sub-system, functions at requirements level or components at design and code level. The approach is supported by a library tool for cataloguing and retrieving reusable components.

The REBOOT (Reuse Based on Object-Oriented Techniques) project is the European reuse initiative within the ESPRIT-2 project which was started in 1990 for four years initially and the research has been carried out in six

countries [Morel and Faget 1993]. The project is based on the faceted classification scheme of components [Sørumgård et al. 1993]. REBOOT applies reuse by composition which means that the reuser builds new systems by composing it from atomic building blocks. REBOOT assumes a vast number of reusable components from different domains are present in a library system. The classification scheme applied here is used for implementing an "intelligent" retrieval mechanism based on keyword-based search mechanism. REBOOT classification is built around four facets; Abstraction, Operations, Operates on and Dependencies. Later in this thesis we will discuss our approach to classify components according to 3-D model of reusable components which are Scope, Behaviour and Reaction. Our model is used for modelling dependencies between components as well as components' retrieval.

Another large-scale reuse programme has been carried out by the Hewlett-Packard company [Lim 1994, Fafchamps 1994]. In HP, the emphasis is on developing a reuse culture within the organisation. Their view to reuse is to divide engineers into two groups; *producers* and *consumers* of work products (code, design, test plans, ...etc.). Producers are creators of reusable work products and consumers are those who use them. The reuse programme included resources to create and maintain reusable work products, a reuse library, reuse tools and implementing reuse-related processes. Fafchamps' research modelled the relationships between producers and consumers and the influence of the organisation structure on the reuse programme. She has identified four models of producer-consumer relationship which are:- *lone producer, nested producer, pool producer* and *team producer*. Each one has its advantages and disadvantages for different organisation structures. Her conclusions were that the team-producer model is the most successful model and the one that provides better cultural shift towards reuse within an organisation because it allows transition from a project to an organisation frame of mind.

2.3 Component-Oriented Reuse

After outlining the characteristics of a reusable component, and the impact of reuse on the development process, it is normal to ask what is a reusable component and how can we find and locate such a component? In the last section some researchers proposed that any work product is potentially reusable [Hutchinson and Hindley 1988; Morel and Faget 1993]. This means a reusable part could be a whole system, a sub-system, functions in the requirements phase or modules in the design and code phase. When we talk about component-oriented reuse (as opposed to generation-based reuse which is outside the scope of this thesis), we mean the process of building new systems from existing building-blocks. The process requires a comprehensive library of reusable components with a scheme for searching and retrieving components from this library.

The research in this area has covered the following aspects of composition-oriented reuse:

- Generic abstract data structures.
- Identifying and locating reusable components.
- Classification schemes.
- Components' interface and interconnection.
- Component retrieval and library management systems.

The work carried out by Booch [1987] is a major effort in the development of generic reusable components. Booch has worked on the domain of abstract data structures and classified them into what he called forms of reusable components. His work was well received because he chose a domain that is well specified and understood by the software development community. His classification of this domain is shown in figure 2-1. He classified reusable components as *Structures, Tools* and *Subsystems.*

Figure 2- 1 Classification of Reusable Components [Booch 1987]

Structures are generic components that denote objects or class of objects characterised as abstract data types. They are classified according to their internal representation as *monolithic* or *polylithic.* The distinction is based on whether the structure contains any sub-structures that can be manipulated independently. For example, a tree is polylithic because it is a recursive component in which it is possible to select part of the tree and treat it as a tree. This distinction is important for the way components are implemented.

Tools are imperative components that act as agents for some algorithmic abstractions that are aimed at an object or class of objects. The relation between objects and tools are close although tools are not objects themselves. Objects are entities, while tools are closely coupled operations (or collection of operations) that act upon entities.

The third type of reusable components that Booch proposed is sub-systems. In such cases, rather than reusing components as building blocks, a collection of components can form a sub-system that can be reused. Booch claims that the higher the abstraction level for reuse, the higher the pay-off gained from reuse and easier to implement.

The other type of classification that Booch proposed is classification according to time and space. Different components could be defined for a single abstraction, depending on its time and space features. These components would look the same from the outside view but vary in the implementation features to suit different applications requirements. Booch calls such variations in a reusable component as *forms* of a component. The same component could come in sequential or concurrent form, bounded or unbounded, managed or unmanaged and iterator or noniterator (or a combination of these forms).

Booch's work was useful and comprehensive for this particular domain, however there are a number of observations that we would like to make about his classification:

- The generic feature of such components is useful for the domain of abstract data structures but is less applicable to other domains whose components cannot be generalised. Therefore, this type of design is better treated as domain-specific design rather than generic reusable components.
- The differentiation between tools and structures in treatment (structures are objects and tools are imperative components) makes it

more difficult for the reuser to comprehend how components are interconnected and used. We believe a unified approach to component presentation and interconnection is crucial for increasing the reusability of a collection of related components.

- The issue of bigger components and higher pay-offs is debatable. If a sub-system (as a bigger-size component) is reused in a particular situation there is no guarantee that the same sub-system would serve the same purpose in another situation without modification. In this case the effort in building a reusable sub-system for that situation may prove to be wasted if no perfect match could be achieved. On the other hand, a bigger component is more difficult to comprehend and interconnect because of its higher degree of complexity.
- The implementation of the component's forms are made as a repeat for the whole internal details. With the use of objects and object-oriented techniques, we believe that polymorphic objects and inheritance would form a better basis for modelling and implementing variations in the implementation details for a collection of related components. In a later chapter of this thesis we will propose another scheme for classifying reusable components to deal with this issue.
- The guidelines and implementation of the components are based on the language he used (Ada), hence many of the design decisions are specific to Ada.

Another work targeting Ada components was presented in [Carter 1990]. This has dealt with concurrency in Ada components and criticised Booch's use of Abstract Data Types (ADT) to implement reusable concurrent components. The criticism is based on the fact that in order to solve some of the problems with concurrency, we have to allow a certain degree of violation to the integrity of the component's abstraction as in the implementation of a binary semaphore (in Booch's guarded form implementation), or the use of shared variables which violates the principles of information hiding and locality. The proposed solution (in

Carter's paper) to this problem is based on the use of Abstract State Machines (ASM) instead of ADTs. The author claims that this solution is easier understood and presented using modern graphical software development methods.

Carter's proposals provide a solution to the issue of violating the information hiding principles in concurrent components but creates another problem to the reuser. The reuser needs to understand additional design concepts to the ones he intends to the required components. This work as well as Booch's approach have shown that solving the technical problems could cause additional problems that discourage reuse. In our approach, a technical solution is proposed which also emphasises how a component is applied when it is reused.

2.3.1 Component Retrieval Mechanisms

The success of reusing software components is bound by the existence of a large library of such components and how they could be identified and retrieved. When a set of requirements is analysed, the first step in development with reuse comprises finding components that satisfy those requirements. When the number of components in the library is large, developers can no longer afford to examine and inspect each component individually to check its suitability. We need an automated method to perform a search and match process to retrieve a list of potentially reusable components.

The existing approaches to component's retrieval cover a wide spectrum of search and matching algorithms. In general they fall into three main streams which are:

1. Text-based retrieval.
2. Lexical descriptor-based retrieval.
3. Specification-based retrieval.

With text-based retrieval, the textual representation of a component is used as an implicit functional descriptor. The users then supply arbitrarily complex string search expressions which are matched against the textual representation. The main advantage of such an approach is related to cost; no encoding is required and queries are fairly easy to formulate. Its disadvantages are simply that plain-text encoding is neither sound nor complete.

Plain-text encoding and search have been used in a number of software libraries alone or in conjunction with other search methods [Frakes and Najmeh 1990; Yoelle et al 1991] and had fairly good recall and precision rates. In a controlled experiment performed at the Software Productivity Consortium. Frakes and Pole found that more sophisticated methods had no provable advantages over plain text retrieval in terms of recall and precision [Frakes and Pole 1992]. However, they found that developers took 60% more time than with the best method to be satisfied that they had retrieved all the items relevant to their queries. This accounts for both the speed with which individual search statements/expressions can be formulated and the number of *distinct* search statements that had to be submitted to answer the same query. With traditional document retrieval systems such as library systems, longer search times are a mere annoyance. In a reuse context, bigger search times can make the difference between reusing and not reusing.

With lexical descriptor-based encoding, each component is assigned a set of *key phrases* that tell what the components offers. Domain experts inspect the components and assign to them key phrases taken from a pre-defined vocabulary that reflects the important concepts in the domain [Burton et al. 1987; Prieto-Diaz and Freeman 1987]. Notwithstanding the possibility of human error and the coarseness of the indexing vocabulary, such encoding is sound, as opposed to plain-text encoding. Further, because a key phrase need not be occurring in the component's textual

description to be assigned to it, it is also more complete than plain text encoding.

Lexical descriptor-based encoding and retrieval suffers from a number of problems. First, an agreed vocabulary has to be developed. That is both labour-intensive and conceptually challenging. In [Sørumgård et al. 1993], a number of problems in developing and using classification vocabulary have been reported. They experienced known problems in building indexing vocabularies for document retrieval, including trade-offs between precision and size of the vocabulary and the choice between what is referred to as pre-conditioned and post-conditioned *indexing*, with the confusion that may result from mixing the two. Software-specific challenges include the fact that one-word or one-phrase abstractions are hard to come by in the software domain.

Further, it is not clear whether indexing should describe the *computational* semantics of a component or its *application* semantics. Characterising computational semantics could help reuse across application domains. However, reusers may have the tendency to formulate their queries in application-meaningful terms. Finally, neither the encoding mechanism nor the retrieval algorithm lend themselves to assessing the effort required to modify a component that does not perfectly match the query.

Specification-based encoding and retrieving comes closest to achieving full equivalence between what a component is and does and how it is encoded. With text and lexical descriptor-based methods, retrieval algorithms treat queries and codes as mere symbols, and any meaning assigned to queries, component codes, and the extent of match between them is external to the encoding language. Further, being natural language-based, the codes are inherently ambiguous and imprecise. By contrast, specification languages have their own semantics within which the fitness of a component to a

query can be formally established [Chen et al. 1993; Mili et al. 1994; Zaremski and Wing 1993]. The formal specification-based methods correspond to what is called partial order-based retrieval, using a partial-order relationship between specifications. This partial order is often used to pre-organise the components of the library to reduce the number of comparisons between specifications.

In [Mili et al. 1994], the authors describe a method for organising and retrieving components that uses relational specifications of programs and refinement ordering between them. Their method is based on two concepts; the first is that there is an ordered relationship between the program specifications such that the program which satisfies a given specification would satisfy the specifications above it. The second is that a specification retrieves the program attached to it as well as those attached to specifications that are below it. Two forms of retrieval are defined: *exact retrieval,* which fetches all the specifications that are more refined than a reuser-supplied specifications, and *approximate retrieval,* which is invoked whenever the exact retrieval fails, and which retrieves specifications that have the biggest overlap with the reuser's specifications. They claim that the approximate retrieval may be useful in suggesting a way of modifying the retrieved programs to make them satisfy the requirements although it does not directly assess the effort required to modify a requirement.

The approach proposed in [Chen et al. 1993] uses algebraic specifications for abstract data types and an implementation partial ordering between them. Reusable components, which may be seen as abstract data types, are specified by both their signature and their *behaviour axioms.* However, while the *implementation* relationship takes into account the behaviour axioms, the retrieval algorithm uses only signatures, which is a renaming of the "types" of the components to match those of the query; the

authors did envisage using an interactive system for algebraic implementation proofs.

Zaremski and Wing propose an approach based exclusively on signature matching [Zaremski and Wing 1993]. The major advantage of their approach is that the information required for matching can be extracted directly from the code. They first define *exact matches* between function signatures, to within parameter names, and then define module signature and partial matches between modules using various generalisation and sub-typing relationships. They too envisage taking into account behavioural specifications in future versions, using LARCH specifications [Guttag et al. 1985], which would then have to be encoded manually.

None of the formal specification-based methods addresses directly the issue of assessing the effort required to modify a component retrieved by approximate retrieval (partial match). Further specification-based methods that include *behavioural* specifications (and not just signatures) suffer from considerable costs. First, there is the cost of deriving and validating formal specifications for the components of the library. This cost is recoverable because it could be amortised over several trouble-free uses of the components and is minimal if specifications are written before the components are implemented. The second cost has to do with the computational complexity of proof procedures. This cost can be reduced if actual proofs are performed only for those components that match a simplified form of the specifications, e.g., the signature; not much can be done about the inherent complexity of proof procedures without sacrificing specification power. The last cost is the cost for the reuser to write comprehensive specifications for the desired components. Because there is no evidence that specifications are either easier or shorter to write than programs, reusers need motivations other than time-savings, or computer assistance, to write specifications for the components they need.

2.3.2 Component-Based Software Development

The availability of a library of software components (no matter how big it is) does not guarantee that software engineers will use (or reuse) them in the development of new systems. We have already outlined the significance of an approach (for development with reusable software) within an organisation before developers could reuse software. As a matter of fact the whole software development life-cycle should be modified to accommodate the new technology. In this section, we will review a number of approaches to developing software systems from software components.

A software life cycle is a model for organising, planning and controlling the activities associated with software development and maintenance [Peters 1987]. For the most part, a life cycle identifies development tasks, elucidates and standardises intermediate deliverables and reviews and evaluates the overall process [Mili et al. 1995]. Existing life cycles may be classified based on the kind of development tasks, deliverables and the organisation of such tasks. For example, the waterfall life cycle, the spiral model [Boehm 1987] and to some extent prototyping, all involve some measure of analysis, design, coding and testing. Nevertheless, whereas the waterfall life cycle implies that an entire system is analysed before any part of it is designed and implemented, both the spiral model and prototyping prescribe the analysis-to-testing cycle on system increments [Agresti 1986].

When we talk about component-based development, existing approaches normally consider the process as two separate life cycles; the life cycle for developing reusable components and the life cycle for developing with reusable components.

In [Sommerville 1992], a reuse-driven approach is proposed, assuming a library of reusable components already exists. In this approach the design

of a component-based system could be modified according to the specifications of existing components. As shown in figure 2-2, the system requirements are modified according to the reusable components available in the library. A side effect of this process is that there may have to be compromises over the system's original requirements and that the design may be less efficient. If, however, a wide-range of component forms is available in the library then the developer can choose from a selection of different versions of a single component that can meet a wide-spectrum of specifications. Still there is a need for an approach that tells the reuser about the different versions of that component and the relationship between them to help him/her find the best possible match between the system's requirements and the components' specifications.

Figure 2- 2 Reuse Driven Software Development [Sommerville 1992]

Hall and Boldyreff proposed a simple model of the reuse process which identifies a number of steps to be taken in proceeding from the recognition of the opportunity for reuse to the actual reuse of components in new applications [Hall and Boldyreff 1991]. Figure 2-3 shows these steps in the context of a component library, using data flow diagram conventions of structured systems analysis and design method (SSADM). There is clear distinction between development for reuse and development with reuse

knowledge should play a more driving role in the process where both phases of the process are inspired by its results.

In [Caldieri and Basili 1991], the authors proposed an approach which mimics the software factory approach (see section 2.2.2). In their model, project teams do no programming (see figure 2-4). They are responsible for requirements and design specifications, which they submit to the *experience factory*, and for integration and testing. The experience factory's activities can be divided into *synchronous activities* and *asynchronous activities*. In the first, activities are initiated following requests from the project teams, and can range from a simple look-up to building the required components from scratch. Such activities are subject to project teams' schedules. The asynchronous activities, on the other hand, consisting of creating components that are likely to be requested (anticipating future demands), or re-engineering components generated by the synchronous activities to enhance their reusability.

Figure 2- 4 Reuse Framework and Organisation[Caldieri and Basili 1991]

The work, presented in [Basili et al. 1992], reported on experiences at the Software Engineering Laboratory (SEL), funded and operated by the University of Maryland, NASA, and the Computer Sciences Corp., in which further research on the above software factory has been pursued. The emphasis here is on the experience factory which was responsible mainly for *process* (vs. *product*) development and reuse [Basili and Green 1994]. Over a period of five years, reuse rates increased from 26% to 96%, the cost per delivered statement decreased by 58%, a 138% increase in productivity - and the number of errors decreased by a factor of four [Basili et al. 1992]. It is not clear how a pure producer-consumer relationship between the experience factory and the project teams would have worked.

The fountain model for object-oriented software development [Henderson-Sellers and Edwards 1993], introduces a different view of the development life cycle, which combines both the incremental nature of the spiral model and a component library. The paper first introduces the fountain model for object-oriented software development on three different levels, which are system level, sub-system level and class development level. Then the model is extended to allow for reuse of classes from a class repository (*software pool*). Figure 2-5 shows the fountain model for object-oriented (O-O) development. As shown in the figure, the model is based on the following principles:

1. There is an overlap between analysis, design and implementation phases with iterative cycles across two (or all three) of these broad phases.
2. The entire model is based on the existence of the '*software pool*' or '*repository*' of classes.
3. There are three development possibilities; new sub-systems, partial reuse/partial modification and total reuse.

4. The concept of domain analysis is integrated with the application life cycle to form a coherent life cycle for an organisation. The system level life cycle uses components discovered by the domain analysis activity and results in components that are generalised for reuse in other development projects.

Figure 2- 5 The Fountain model for object-oriented development process [Henderson-Sellers and Edwards 1993]

This model provides a useful analogy to describe the O-O life cycle, however some of the important issues in the process have been dealt with casually. For example, it has not addressed domain analysis and domain knowledge in details. Domain analysis was superimposed on the model without a clear idea about its role or its deliverables. Furthermore, there is no indication how classes are identified from the pool and assessed for reuse within the new applications.

2.4 Domain Analysis

Domain analysis (DA) was first introduced by Neighbors [1984] as a process of "identifying the objects and operations of a class of similar systems in a particular problem domain". Neighbors views domain analysis with analogy to systems analysis such that system analysis deals with the specifications in a specific system, while domain analysis describes the common actions and objects in all systems in an application area. Domain analysis can be performed prior to systems analysis and its output (domain model) supports systems analysis in the same way that the systems analysis output (specifications document) supports the system design, see figure 2-6.

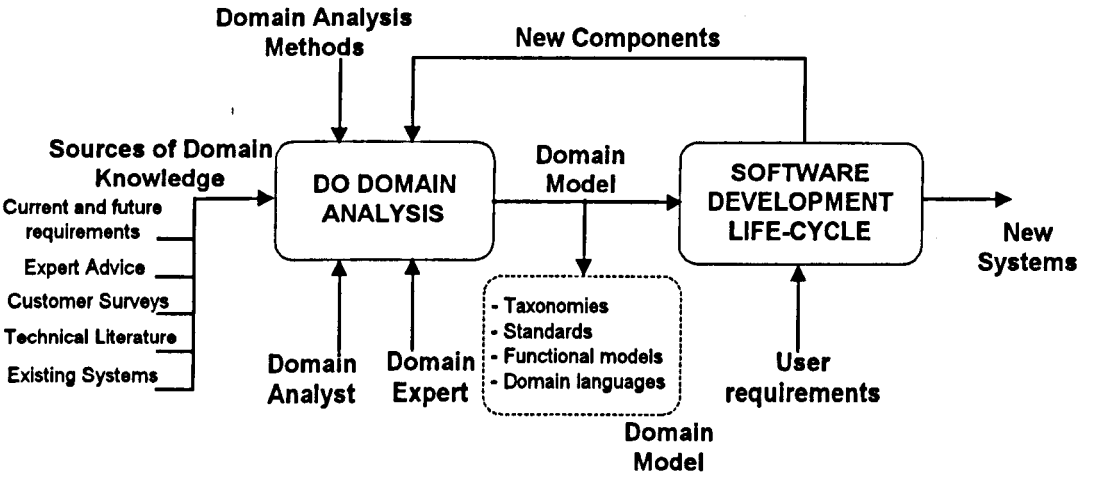

Figure 2- 6 Domain Analysis and Domain Model

Domain analysis represents a higher level of abstraction than systems analysis. In the conventional water-fall model, systems analysis incorporates creating a model of the system with suggestions to automate or improve this system. The outcome of systems analysis is then used by the system designer to produce a particular design that meets a set of requirements and specifications. The two activities (requirements analysis and system design) deal with a model of a particular system. In domain

analysis, on the other hand, a domain model is created in which all systems in a specific application are generalised. Domain analysis includes generalising common characteristics from similar systems, identifying objects and operations common to all systems and defining a model that describes the relationship between them.

2.4.1 Domain Analysis Process

Domain analysis is actually an information collecting, analysing and presenting process about a specific application domain. Figure 2-6 shows the inputs, outputs and agents that comprise the elements of a domain analysis process [Prieto-Diaz 1990]. In general, different approaches to domain analysis agree about the inputs, outputs and agents of the process but they provide alternative ways to realise the outputs from the inputs. Two agents are required for the domain analysis process which are a domain expert and a domain analyst. The domain expert is a person who is familiar with the application area and who does not have to be a systems analyst or a software engineer. His role is vital for identifying the relevant areas in the domain and the relationship between the different objects and functions in the domain. In many cases the domain expert's role is to verify and organise the information acquired from other sources of knowledge. The other agent of the process, the domain analyst, is a person who is responsible for collecting and analysing the information about the domain from the different sources and present these information in a domain model according to a domain analysis method.

The inputs and outputs of the process are:

2.4.1.1 Inputs

The inputs to the process are the sources from which information is acquired. These sources provide knowledge about a problem domain and the models for implementing software-intensive solutions to problems. The inputs to the process include:

1. Technical literature: Textbooks, scientific journals and manuals.
2. Existing applications, which can be investigated such as source code, design documentation, user manuals and the results of reverse-engineering the implementations.
3. Customer surveys and market analysis.
4. Human expertise in the problem domain (e.g. the expertise of accountants, chemists, or police officers) and the design of systems in that domain (The expertise of systems analysts, designers, programmers or maintainers).
5. Historical records of evolution in the domain.

For practical domain analysis, each source of information has advantages and limitations. Human experts are good sources for creating general views of the conceptual structure of a problem domain. These views help the analysts understand the interaction between the mass of information to be examined during the process of domain analysis. Human experts are usually the only sources for justifications or explanation of the system's way of operation. A domain expert's memory is usually rich with historical information that cannot be found any where else. Nevertheless, the time of human experts is usually scarce and costly.

The technical literature often provide precise and detailed data but it is not likely that it contains insights and causal or historical knowledge. Although this source of information is cheap and available, it cannot be credited for insights, justifications or elaboration . Existing applications are useful as practical examples for the knowledge required from experts. They help by clarifying and discovering information like variations in the definitions of domain objects, relations, constraints, specialised design plans and implementation knowledge. However, they are very specific and it is time consuming to move from a specific application abstraction to a new one. Market surveys do not provide much more than statistical distributions of market needs. However, they provide pragmatic grounds for establishing whether specific properties are essential, common or rare.

The various information sources can act together to provide the analyst with a clear picture of the domain rather than using only one or two sources of information.

2.4.1.2 Outputs:

The output of the domain analysis process is a model of the domain. The contents of the model are determined by the requirements of the software construction process. Thus, a useful model for an application domain should contain at least:

- A definition of the concepts used in the specification of problems and software systems.
- A definition of what constitutes typical software designs, alternatives, trade-offs, and justifications.
- Software implementation plans

Different models are produced as outputs of the domain analysis process to serve different purposes. A taxonomy model represents a definition model that shows the domain context and its organisation. Knowledge representation models like semantics networks and frames provide domain semantic and some explanatory capabilities. Domain-specific languages are models that may support direct translation of software specification into executable code.

Other models provide information that help in describing the domain. These models may take the form of standards, templates or interface definitions. Functional models provide descriptions about systems operation using graphical representations like data flow diagrams.

2.4.2 Survey of Existing Domain Analysis Methods

In this survey, five approaches to domain analysis are described. In the fifth, we review two object-oriented domain analysis methods. The process, inputs and outputs of each one are illustrated and the advantages and problems encountered in each one are outlined in section 2.4.3.

2.4.2.1 The Draco Approach.

This approach was introduced by Neighbors [1980] and it was the first time that the concept of domain analysis for reuse was mentioned as an activity that generalises the solutions to problems over an application level. Draco's approach has set formal procedures for developing reusable software components on the organisation level [Neighbors 1989]. That means it provides an engineering view for reuse of design in addition to the generation of reusable software. The idea conveyed by the Draco approach is once the domain analysis has been carried out for the application domain then its outcome can be reused for all the systems in that domain.

Figure 2- 7 The Draco Approach [Neighbors 1984]

In the Draco approach, three new human roles have been introduced (as shown in figure 2-7): the application domain analyst, the modelling domain analyst and the domain designer. The application domain analyst defines the objects and operations which can be identified in a class of similar system according to his previous experience and by interacting with users of these systems. His function is compared to the systems

analyst function but over an application area or a domain. The output is a description of all the objects and operations in that domain which is given to the domain designer. The domain designer specifies different implementations for objects and operations using notations of domains already known to Draco. The modelling domain analyst function is similar to the function of the application domain analyst, but is more concerned with which notations and techniques have been successful in modelling a wide range of applications.

The domains are specified to the Draco system by six parts which are: parser, prettyprinter, optimisations, components, generators and analysers. The parser description checks the validity of notations used by Draco to define and manipulate the internal form of a domain. This information can be used to allow or restrict use of inter-domain definitions. The prettyprinter description specifies the external syntax of the domain to be produced by Draco. This enables Draco to interact with the users in the language of the domain. Optimisations are the rules of exchange between the objects and operations within a certain domain. The output of optimising any fragment of the domain language is checked by the parser descriptions, the final arbiter of a well-formed notation fragment in the domain. The semantics of the domain are specified within the components. There is a component for each object or operation in the domain. Different implementations and implementation decisions for each object or operation in terms of one or more refinements are included in the components. The next descriptions are the generators. They are used in some cases, where domains are specified in the form of algorithmic knowledge, to generate domain-specific code. The generators do not do any optimisation tasks but write new codes in the domain. The generators operate and produce internal form of the domain where they were introduced and they are subject to checking by the parser description. The final descriptors are the analysers like data flow analysers, execution monitors, theorem provers and design quality measures, which

manipulate information about an input instance of domain notation. As with all domain-specific procedures, the data produced and consumed by the analysers are kept within the schema described by the domain parser definition.

The Draco system function is to generate a domain specific language using information given by the domain analysis process. Once a statement in a domain language has been parsed into internal form it may be

- prettyprinted back into the external syntax of the domain;
- optimised into a statement in the same domain language;
- taken as input to a program generator that restates the problem in the same domain;
- analysed for possible leads for optimisation, generation or refinement or
- implemented by software components, each of which contains multiple refinements and which make implementation decisions by restating the problem in other domain languages

2.4.2.2 The Prieto-Díaz Approach.

The Prieto-Díaz approach to domain analysis [Prieto-Díaz 1987] (as shown in figure 2-8) is centred around two key actors in the process which are a domain analyst who has the procedural know-how on domain analysis and a domain expert who provides relevant knowledge about the domain in accordance with a set of guidelines. Information can also be extracted from existing systems.

Figure 2- 8 Context Diagram of Prieto-Díaz Approach [Prieto-Díaz 1987]

The domain analysis activities in Prieto-Díaz approach are presented in data flow diagrams and structured in levels of abstractions. In the highest level, three stages are identified - as shown in figure 2-9, which are (1) prepare domain information which contains activities prior to domain analysis (named by Prieto-Díaz as pre-DA), (2) analyse domain and (3) produce reusable work products which are referred to as post-DA according to Prieto-Díaz. Some of the important intermediate outputs can be noticed in this level like DA requirements document, a domain taxonomy and domain frames. A domain model and a domain language are optional products because not all domains can be modelled.

Figure 2- 9 Data Flow Diagram of Domain Analysis Stages [Prieto-Díaz 1987]

In pre-DA stage, a set of DA guidelines are applied to a particular domain. It is the job of the analyst at this stage to define the domain and identify its boundaries. The product of this stage is the DA requirements document. This document should include a high level breakdown of activities in the domain, which part of the domain to analyse, potential areas to modularise, standard examples of available systems, and any issues relevant to that domain.

In the activity of analysing domains, the analyst identifies reusable objects and operations, abstractions and classification. The output of this stage are domain frames and taxonomies.

In post-DA stage, domain frames, taxonomy and a possible domain model are used to produce work products. This involves encapsulating elements that could be candidates for reuse, defining guidelines for reusing individual components and setting standards for building systems in the domain.

2.4.2.3 The IDeA Approach.

This approach is presented by [Lubars 1991] where the process of analysing domains for reuse is distinguished from the process of building the reusable artefacts. The former is called *domain analysis* and the second is *domain engineering*. Analysing a domain for IDeA is a bottom-up process starting from analysing similar problems in an application domain and different solution alternatives to the problems to generalising the solution domain over a number of related application domains. The process incorporates three stages. Each stage results in identifying common abstractions relevant to that stage:

1. Analysis of similar problem solutions. The results are characterisations of solutions of particular classes of problems in the application domain.
2. Analysis of solutions in an application domain. In this stage, the characterisations from stage 1 are grouped to produce characterisations of a particular application domain.
3. Analysis of an abstract application domain. The characterisations defined in stage 2 are generalised in this stage to model related application domain classes.

IDeA concentrates on the reuse of abstract software designs which are represented in the form of design schemas. The design schemas present the designer with solutions to the similar problems in the application domain. The emphasis is on the commonality in the domain leaving variation to be informally specified during the stage of analysing the domains. In addition to the design schemas, the outputs of domain analysis within IDeA includes properties of the objects in the domain, data types in the form of type hierarchy, type constraints and a set of rules for schemas specialisation and refinement.

2.4.2.4 The KAPTUR Approach.

KAPTUR [Moore and Bailin 1991] is a bottom-up solution-oriented approach to domain analysis by going from analysing solutions to specific

problems in an application to generalising the solution scope over the whole domain. The aim of the approach is capturing design decisions and rationales of systems while they are being developed to support the reuse of software assets. These assets are stored in a knowledge base whose creation is time consuming and involves several refinement stages. Models of existing system in the domain are first examined and their common formats are modelled using data flow, entity-relationship and state transition diagrams. Generic models of domain specific variations in the architectural designs of the systems are built using the results from the analysis of the different system designs. These models are then verified by consultations with domain experts to solve the problems concerning the features and operations that are specific to the domain. Experts also help to fill in gaps that may arise in the process building the generic models. Results from consultations with domain experts are used to identify reusable assets that are added to the domain knowledge base.

The outputs of this approach are called assets for reuse and represent the domain knowledge as a framework for reuse [Moore and Bailin 1991]. The framework constitutes tools and products that are used to support reuse in the domain; in general these are:-

- Dialogue-based specification, which are alternative reusable options and their problems and trade-offs.
- Reuse database; these are the products that resulted in the domain analysis process.
- Graphical programming, which are a means of integrating reusable components into new systems.
- Domain-specific very high level language; these are high level abstractions and macros provided to reduce the amount of new code necessary to implement new requirements.

2.4.2.5 Object-Oriented Domain Analysis Methods

The realisation of the object-oriented benefits to software reuse has led to a shift in the concept of problem analysis. Because objects are entities that inherently bear domain features, domain analysis is gradually gaining momentum in the object-oriented analysis methods as an important phase. Capturing the domain specific features in the object design is becoming essential for successful and evolving objects, and in consequence increasing their reusability. This notion (domain analysis in object identification and design) has already been outlined when we discussed the fountain model in section 2.3.2 of this chapter. In this section, we discuss two domain analysis approaches based on the object-oriented technology.

The Shlaer and Mellor method relies on the concept of a domain or a "subject matter" [Shlaer and Mellor 1993]. They claim that thinking of a system development in terms of domains allows for more realistic, multilevel views of the problem as a whole, as well as supporting the object-oriented goal of reuse. This is opposed to the traditional approach of separating problem from solution which, they think, is overly simplistic. The object-oriented domain analysis method proposed by Shlaer and Mellor concentrates on capturing domain-specific knowledge with the help of a domain expert and mapping the knowledge into an object-oriented design.

The approach is based on building three types of 'formal models': *Information Models, State Models* and *Process Models* [Shlaer and Mellor 1989]. In the information models, conceptual entities (objects, attributes and relationships) of the problem are identified and formalised in objects and attributes. Emphasis is placed on formalising the relationships between objects. The state models are used to formalise the "life-cycles" or "life histories" of objects and relationships. The information models describe the static characteristics of objects, while the state models

describe their dynamic behaviour. When a state model is built to describe an object's life-cycle, the behaviour of a single typical, but unspecified instance is formalised. The authors state that single state model suffices to explain the behaviour of all instances, and is analogous to pure code. In the third type of models, the processes required to drive an object or relationship through its life-cycle are derived from the actions of the state models. A separate data flow diagram is constructed for each state in each state model. The data flow diagram for a state depicts, in a graphical form, the process associated with that state.

Reuse (in this method) is achieved through the process models. Shlaer and Mellor express their view of reuse by urging the analyst to compare the data flow diagrams for all the states of a single state model looking for similar processes which are being used repetitively.

Another approach based on object-oriented technology was presented in [Gomaa et al. 1989; Gomaa 1992]; the Evolutionary Domain Life Cycle (EDLC) Model. This is a software life cycle that allows systems to evolve through several iterations eliminating the distinction between software development and maintenance. According to the EDLC model, the traditional system development activities (Requirements Analysis, Requirements Specifications and System Design) are replaced with *domain analysis, domain specification* and *domain design* generating a *domain model* as a deliverable. The domain model is a problem-oriented architecture for the application domain that reflects the similarities and variations of the members of the domain (systems within the domain).

The EDLC consists of three major activities which are *Domain Modelling, Target System Generation* and *Target System Configuration* (see figure 2-10). In domain modelling domain-specific reusable components (reusable specification and reusable architecture) are developed and stored in a reuse library. In target system generation, given the requirements of an individual target system, system specification is generated by tailoring the reusable specification and the target system architecture. In the last activity an instance of the target system is composed based on the target system configuration data.

Figure 2- 10 Evolutionary Domain Life-Cycle Model [Gomaa 1993a]

EDLC models a problem domain by considering similarities and variations among its members. Those objects and features that are common to all members of the domain are called the *kernel* of the domain. Modelling the domain is an iterative process, so that kernel requirements and objects are considered before the variations. The variations in the domain represent iterations on an evolving domain model. Furthermore EDLC identify three types of objects in the model which are: kernel, optional or variants. Kernel objects are those required to satisfy kernel requirements.

2.4.3 Discussion.

Following the review of some domain analysis methods, most are bottom-up approaches where individual problems and their alternative solutions are first analysed and then solutions are extended and generalised over the whole domain. We find this in IDeA, KAPTUR and Draco Approach. In Prieto-Díaz approach, the process is done in what is called a top-down-bottom-up way; hence it is sometimes called the 'sandwich' approach. The domains are defined and their boundaries are identified first (top-down), and then common problems are analysed for reuse in a later stage (bottom-up).

Domain analysis approaches can be classified according to their final products as either language-based or model-based approaches. In the first type, the output is a domain specific language where it is used to specify components within the domain. In some cases, like the Draco approach, they work like application generators. In the second type, the output is a framework for reuse in the domain in a form of a domain model. Typical contents of the model (called assets or artefacts) are: reusable components, reuse guidelines, domain standards.. etc. A typical example of this type is the IDeA approach.

In general, domain analysis activities are conducted on an ad-hoc basis. We cannot point out a systematic approach to carry out domain analysis activities in a well-defined fashion. Although the outputs are defined in all approaches, it is not clear how these outputs are going to be utilised in the process of developing new systems from reusable components. This problem is more obvious in the model-based approaches.

Nevertheless, domain knowledge acquisition is the basis for all the approaches. They all recognised the importance of capturing domain knowledge prior to developing reusable components and therefore domain analysis is regarded as a part of the development for reuse process. They

may, however, vary in the timing of carrying out domain analysis; some view it as a front-end stage prior to actual software development, while others regard it as an integrated part of the software development for reuse. A detailed comparison of domain analysis methods was presented in [Watrik and Prieto-Diaz 1992].

In comparison, the object-oriented domain analysis methods are more focused since they all consider objects as the foundations of the development process. This leads to the identification of domain-specific features within objects at very early stages of the analysis process. In the Shlaer and Mellor method for domain analysis, reuse is opportunistic rather than systematic. This is a feature that most domain analysis methods seem to have. The EDLC model, on the other hand, deals with reuse more systematically. Reusable objects are identified early in the life cycle and distinguished from other types of objects. However, this approach is aimed at developing distributed applications in which many of its features are relevant to this domain. The approach also sets criteria for identifying objects from domain requirements and multiple views to domain modelling. These criteria are mainly relevant to the domain of distributed applications. The approach is clearly more reuse-oriented than other object-oriented analysis methods but it needs extension to make it applicable to other application domains.

2.5 Summary

The discussion in this chapter has highlighted a number of issues which need to be addressed for increasing the likelihood of reusing software components. In chapters 4-7, a new approach for reusing domain-oriented components is proposed. This approach attempts to address the following points related to reuse:-

- One of the main obstacles to software reuse is component's comprehension. Reusers need to have a clear idea about what the

component can offer their application and how to integrate it with other parts of the system.

- Linked to the previous point, and as was pointed out by [Biggerstaff and Ritcher 1987], the presentation factor is very important to understand components abstraction and scope. We need an organised way to present the domain knowledge and domain analysis results before we could make effective use of its resources.
- Some cultural resistance among practitioners to adopt reuse within organisations which, in our view, is attributed to a lack in the technical support that facilitates the re-application of components in a systematic fashion. There is real need for a technical approach that transforms the reuse process from an ad-hoc to a systematic approach.
- Component retrieval mechanisms needs to stress the idea of how a component fits with other components in a certain application. A retrieval mechanism should also be an integrated part of the entire development process.
- This also means that the software development life-cycle need to be changed to accommodate domain-analysis, component retrieval and reuse assessment.

The proposed approach is an attempt to solve the above points through providing a technical support to manage reusable assets as well as a process for development with reuse based on domain analysis. This is done by introducing a new technology to model the components and their relationships within specific domain scope. The results of the domain analysis are presented in a structural way which enables effective identification and retrieval of the reusable assets when systems are synthesised. A number of design guidelines are introduced for applying the approach in software development. These are general guidelines that are not language-specific.

Chapter Three

3. Object-Oriented Design Patterns and Software Architectures

3.1 Introduction

A critical aspect of the design for any large software system is its high-level organisation of computational elements and interactions between those elements. Broadly speaking, this is the software architectural level of design. Recently software architecture has begun to emerge as an explicit field of study for software engineering practitioners and researchers [Garlan 1995; Garlan and Perry 1995]. There is a large body of recent work in areas such as module interface languages, domain-specific architectures, architectural description languages, design patterns and pattern catalogues and architectural design environments.

In this chapter, we will review some of the work in the areas of software architecture and object-oriented design (OOD) patterns. First, the support of design patterns to software reuse is discussed and then the concept of design patterns and its engineering background is outlined in section 3.3.2.1. In section 3.3.2.2, some common patterns in object-oriented design are described. In this study, we introduce some examples to illustrate how design patterns solve particular design problems in OOD, and to highlight the differences between design patterns and software architectures. The purpose of this review is to underline how design patterns differ from the generic software architectures that are explained in chapter 5.

In 3.3.3, we discuss software architectures and start by describing four common architectural styles. Then in section 3.3.3.2, we discuss what a domain specific software architecture is and in section 3.3.3.3, we review some of the research efforts in the area of software architecture

3.2 Object-Oriented Design Patterns and Software Reuse

Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them. Your design should be specific to the problem at hand but also general enough to address future problems and requirements. You also want to avoid redesign, or at least minimise it. Experienced object-oriented designers already know that a reusable and flexible design is difficult if not impossible to get right the first time. They usually try to reuse it several times, modifying it each time until the design is matured.

Identifying reusable and flexible designs is part of the experience that a designer gains with time. Usually good solutions are used in the development of similar systems and in turn increasing the reliability of the resulting systems. Consequently, patterns of classes and communicating objects will be found in many object-oriented systems. These patterns solve specific design problems and make object-oriented designs more flexible and ultimately reusable.

Design patterns (sometimes referred to as micro-architectures) make it easier to reuse successful designs and architectures. Building software architectures in terms of known patterns will result in a higher level system design (software architecture) that is easier to build, more reusable and simply mapped into detailed design and code [Gamma et al. 1995].

The study of design patterns is gaining more attention for its anticipated influence on the software industry [Booch 1993; Dutto and Sims 1994; Johnson

1994; Coad 1992]. Design patterns, in conjunction with related subjects such as frameworks and architectures, could dramatically change the way software will be designed and written.

3.2.1 What Is a Design Pattern?

In order to identify and use (or reuse) patterns among object-oriented designs successfully, we need to explain what a design pattern is, what the elements of design patterns are and what the uses of design patterns are.

Design patterns is not a new concept in other engineering disciplines. In building engineering, for examples, patterns have been used as a way for reusing experience in the design of buildings and towns. Christopher Alexander, an architect, says, "each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [Alexander et al. 1977]. Even though the above quote is extracted from building engineering, the concept is true for object-oriented design. Gamma et al. define a design pattern as "a mechanism for expressing design structures. Design patterns identify, name and abstract common themes in object-oriented design" [Gamma et al. 1993]. They have conducted a research on object-oriented designs and frameworks and observed that there exists idiomatic class and object structures that help make designs more flexible, reusable and elegant. For example, the *Model-View-Controller* (MVC) paradigm from Smalltalk is a design structure that separates representation from presentation. MVC promotes flexibility in the choice of views, independent of the model.

Peter Coad puts a definition to object-oriented design patterns as follows:

> *An object-oriented pattern is an abstraction of a doublet, triplet, or other small grouping of classes that is likely to be helpful again and again in object-oriented development.* [Coad 1992].

Erich Gamma identifies three essential parts that constitute a design pattern [Gamma et al. 1993]:

1. An abstract description of a class or object collaboration and its structure. The description is abstract because it concerns abstract design, not a particular design.
2. The issue in system design is addressed by the abstract structure. This determines the circumstances in which the design pattern is applicable.
3. The consequences of applying the abstract structure to a system's architecture. These determine if the pattern should be applied in view of other design constraints.

Gamma also identifies a number of uses of patterns in the object-oriented development process[Gamma et al. 1993]:

- design patterns provide a common vocabulary for designers to communicate, document and explore design alternatives.
- Design patterns constitute a reusable base of experience for building reusable software. They extract and provide a means to reuse the design knowledge gained by experienced practitioners.
- Design patterns act as building blocks for constructing more complex designs; they can be considered as *micro-architectures* that contribute to overall system architecture.
- Design patterns help reduce the learning time for a class library. Once a library consumer has learned the design patterns in one library, he can reuse this experience when learning a new library.
- Design patterns provide a target for the recognition or re-factoring of class hierarchy.

3.2.2 Some Common Patterns

Some research work is currently going on identifying and cataloguing design patterns (see Booch 1993). Erich Gamma's book on design patterns [Gamma et al. 1995] contains a catalogue of patterns that are organised according to **scope** and **purpose** as shown in Table 3-1

According to Gamma's classification of patterns with respect to purpose, patterns can be either **creational**, **structural**, or **behavioural**. Creational patterns concern the purpose of object creation. Structural patterns deal with the composition of classes or objects. Behavioural patterns characterise the ways in which classes or objects interact and distribute responsibility.

Table 3-1 Design pattern space [Gamma et al. 1995].

With respect to scope, Gamma's classification specifies whether the pattern applies primarily to classes or to objects. The main difference between class and object scope is in that *class patterns* are concerned with the organisation of the inheritance (class-subclass) relationship; while *object patterns* deal with the organisation of object collaboration. The general features of each category of patterns are shown in Table 3-2

We have selected three patterns from Gamma's catalogue as examples of the three categories (creational, structural and behavioural). The choice of these patterns (Factory Method, Adapter and Strategy) is made for their common use in the design of object-oriented software.

Table 3-2 General features of design patterns

		Purpose		
		Creational	**Structural**	**Behavioural**
Scope	**Class**	defer some part of object creation to subclasses.	use inheritance to compose classes	use inheritance to describe algorithms and flow of control
	Object	defer some part of object creation to another object	describe ways to assemble objects	describe how a group of objects cooperate to perform a task that no single object can carry out alone

The first pattern is *Factory Method* which is a class creational pattern. In this pattern the intent is to define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. This is a common design decision that happens almost in every object-oriented system, which may be best presented by an example. Consider an application for presenting multiple documents to the user. There are two abstract classes (an abstract class is a class that cannot have an object instantiated from it) in this scenario, which are *Application* and *Document*, and *Application* is responsible for managing *Document* objects. To create a particular application type, you need a particular type of documents (e.g. drawing application and drawing document). The abstract class *Application* knows that it needs to create a *Document* and when to create it. However because the particular *Document* subclass to instantiate is application-specific, *Application* does not know what type of *Document* to instantiate. This causes a problem: A class must instantiate other classes, but it only knows about abstract classes, which it cannot instantiate [Gamma et al. 1995].

The Factory Method pattern offers a solution. It encapsulates the knowledge of which *Document* subclass to create and provides an abstract *CreateDocument* method that simulates the creation of the document (see Figure 3-1). The actual creation of *Document* is deferred to *Application's* subclasses, which redefine the method *CreateDocument* where the appropriate *Document* subclass is created. Thus, the interface of *Application* abstract class could be used at compile time without having to worry about the type of the *Document* object. At run time though, the relevant object of *Application* (*MyApplication*) is in a position to create the needed object whose type (*MyDocument*) is a subclass of *Document*. This object could even be manipulated by the methods of the abstract class *Application*, since its parent class *Document* is already known to *Application*.

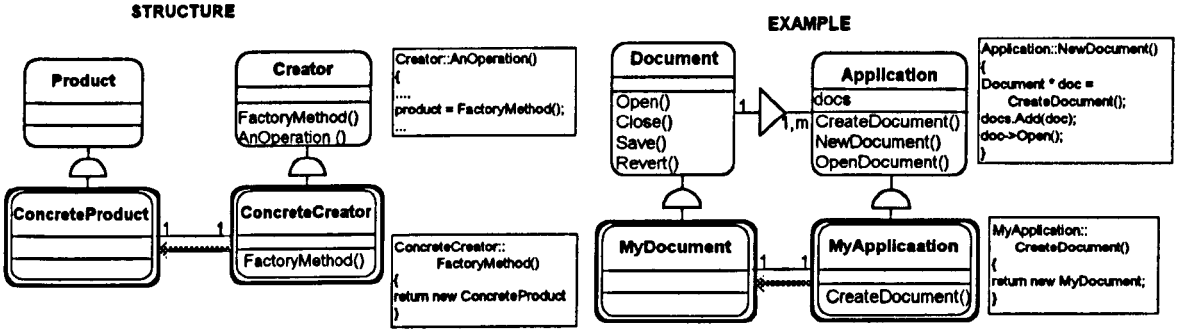

Figure 3-1 Factory Method Pattern

The second pattern is *Adapter*, which is both a class and an object structural pattern. The intent is to convert the interface of a class into another interface that clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces. This is another common situation in object-oriented design where a particular class cannot be reused because of its unfamiliar interface. The Adapter pattern resolves the situation by introducing a new class (usually through inheritance) that conforms the two interfaces.

Let us consider the example of a drawing editor that lets users draw and arrange graphical elements (lines, polygons, text, etc.) into pictures and diagrams. The drawing editor's key abstraction is the graphical object, which has an editable shape and can draw itself. The interface for graphical objects is defined by an

abstract class called Shape. The editor defines a subclass of Shape for each kind of graphical object: a LineShape class for lines, a PolygonShape class for Polygons, and so forth [Gamma et al. 1995].

The graphical editor example represents a typical example of an application that incorporates objects that behave expectedly (LineShape and PolygonShape), and have a familiar interface, and an object (TextShape) that has a more complicated behaviour. Nevertheless the graphical editor needs to treat all of them in like manner. If an off-the-shelf object is to be reused for providing a complete and sophisticated text handling facility (TextView), the situation is that it is very likely that TextView will have an interface that is not compatible to Shape. This means we cannot use TextView and Shape objects interchangeably.

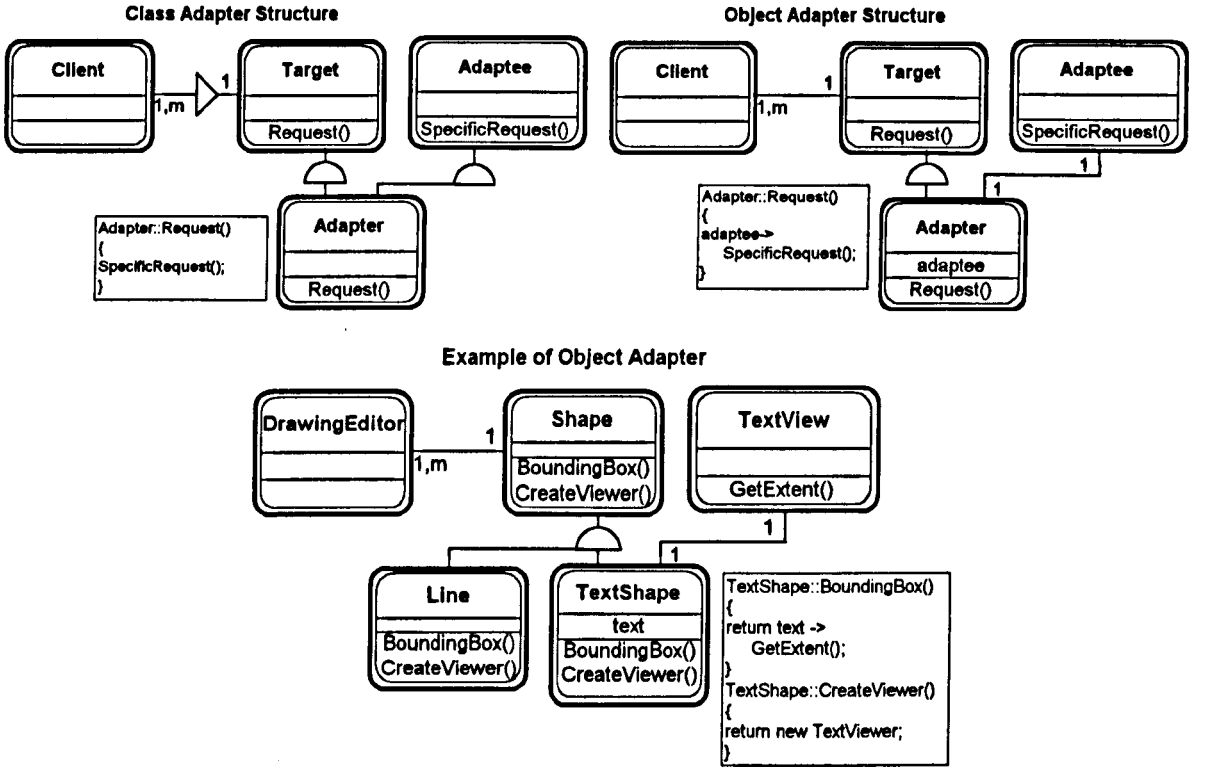

Figure 3-2 Adapter Pattern

The *Adapter* pattern (see Figure 3-2) provides a mechanism for adapting the TextView interface to Shape's. This pattern comes in two forms; one is class structural form using multiple inheritance for inheriting Shape's interface and

TextView's implementation, and the second is object structural form by composing a TextView instance within a TextShape and implementing TextShape in terms of TextView's interface.

The third commonly used pattern is *Strategy*, which is an object behavioural pattern. The intent of the pattern is to define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. This pattern can be applied in situations when there are a number of related classes that differ only in their behaviour, or there are a number of possible alternative implementations of a certain algorithm within a certain abstraction.

Consider the example of breaking a stream of text into lines, many algorithms exist for doing this. Hard-coding all such algorithms into the classes is not desirable because the classes that utilise them get more complex, different algorithms will be appropriate in different situations and it is too difficult to add new algorithms and vary existing ones when line breaking is an integral part of a client. These problems can be avoided by defining classes that encapsulate different line breaking algorithms. An algorithm that is encapsulated this way is called a **strategy** [Gamma et al. 1995].

As seen in Figure 3-3, suppose a Composition class is responsible for maintaining and updating the linebreaks of text displayed in a text viewer. The actual implementation of line breaking strategies is done separately in a subclasses instead of implementing them within Composition.

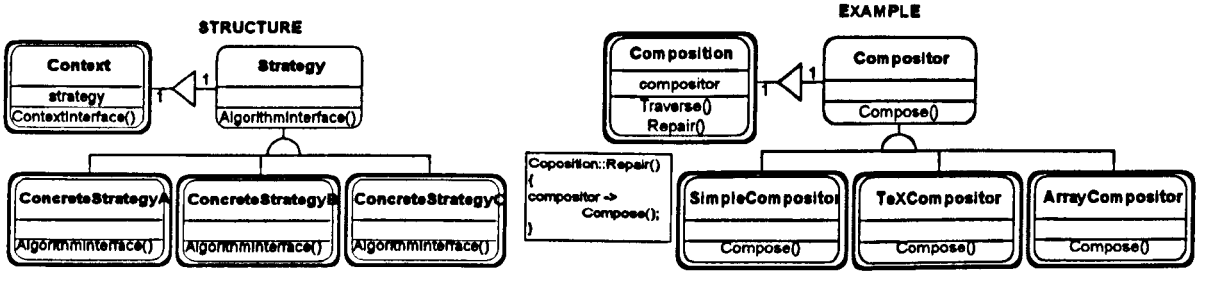

Figure 3-3 Strategy Pattern

In many cases, it may seem (for experienced designers) that Gamma is stating the obvious when he describes his patterns. However, for a novice designer, the patterns provide insights for good (as well as reusable) object-oriented design. It is also obvious that inheritance and polymorphism are the corner stones of Gamma's patterns in most of his patterns. This does not come surprisingly to object-oriented practitioners as these are the features (in addition to encapsulation and abstraction) that distinguish this technology and make it more supportive to reuse.

Another set of patterns was presented by Peter Coad [Coad 1992]. Some of the patterns are the same as Gamma's patterns but having different names. As an example, the Strategy pattern is introduced by Coad as Roles-played pattern. We have chosen one pattern from Coad's set. Figure 3-4 shows the structure and an example of the pattern called *Broadcast* (known as Observer in Gamma's catalogue). This pattern is used to communicate complex changes between one major section of an OOA/OOD model with another major section. Whenever it changes, a "broadcasting item" object broadcasts a change notification to the "receiving item" objects that it knows about. A notified "receiving item" object then sends a message to the "broadcasting item" object to get the change; once it gets the change, a "receiving item" object takes whatever action is necessary in light of the change.

Figure 3-4 Broadcast Pattern [Coad 1992]

As an example, the pattern is applied to keep human interaction distinct from business domain classes. This is done to simplify both parts; and it is done to increase the likelihood of reuse for each part. A "human interaction view" object gets user input and sends a message to invoke action to the corresponding "model" object. At some point in time, when a change does occur, a "model" object" broadcasts a change notification to its dependent" human interaction view" objects. Then each dependent "human interaction view" object sends a message to get the change; on receipt of the change, the "human interaction view" updates its display (see Figure 3-4).

Bruce Anderson has catalysed significant study into the codification of patterns. He also established workshops (during OOPSLA conferences) focused on the creation of an architecture handbook the purpose of which is ultimately to serve as a catalogue of patterns [Anderson 1994]. His catalogue of patterns contained a number of patterns that have been proposed by the participants of the workshops with brief description of each pattern.

Patterns within object-oriented designs are important tools for getting the most of your design and represent a higher-leverage form of reuse. The search for patterns encompasses far more than finding perfect class abstraction; rather, it focuses upon identifying the common behaviour and interactions that transcend individual objects. Nevertheless, a number of different sets of patterns might confuse their users where they are supposed to provide common ground (between practitioners) for communication and documentation; especially when the same pattern is named different names in different catalogues. One unified set of patterns could be very useful for designers of object-oriented software.

3.3 Software Architectures

Recent research works have shown that some of the problems encountered in engineering the reuse process could be solved by considering the software architectures in a domain [Kogut and Clements 1994; Tracz 1994; Garlan et al. 1995; Shaw 1995]. Software architecture is an organisational structure of a

system that includes components, connections, constraints and rationale [Garlan and Shaw 1993]. Software architectures form a higher level of system design that involves decisions made early in the life-cycle. These decisions have their impact on the way the systems are analysed and designed. The whole life-cycle could then be driven by the architectural style, hence providing a framework for designing and reusing the components and their connections.

Software architecture is concerned with design at the system level. Certainly this includes system structure (or topology), discriminations among different kinds of structures, and abstractions or generalisations about structures and families of similar structures. It also includes identification, specification, and analysis of the properties that are related to these structures, either because they influence the selection of a structure or because they are consequences of that structure.

At the architecture level, the components of interest are modules and the interconnections among modules. Architectural styles guide the selection of kinds of components and of the strategies for composing them. As a result, the kinds of components and interconnections can differ substantially between architectural styles. The properties of interest include system structure, gross performance, component consistency, and other aggregate properties such as security and reliability.

The efforts for engineering the software architectures have focused on the identification and modelling of architectural styles for designing software systems in a domain [Shaw 1995, Kruchten 1995]. Some of the attempts comprised the use of architecture description languages [Kazman and Bass 1994]. However these attempts focused, in general, on the linkage between sub-systems rather than between components and modules. Some of the work considered the use of building blocks in creating system's architectures [Van Der Linden and Muller 1995]. In their paper, they proposed a method for building sound architectures for large-system development by decomposing the system

into building blocks (hardware and software) in order to decrease system complexity.

3.3.1 Architectural Styles

Software architecture is an emerging field whose theory is still not fully-developed and its taxonomy is not well-accepted [Garlan and Shaw 1993]. However, we can now identify a number of architectural patterns, or styles, that currently form the basic repertoire of a software architect. Typically a software system involve some combination of several styles.

Gralan and Shaw have considered a number of common architectural styles upon which many existing systems are currently based [Garlan and Shaw 1993]. Many of these styles are extracted from existing systems in practice and how they are organised. The choice of an architecture style in a system may considerably affect design decisions in the design of the system. In her paper [Shaw 1994], Mary Shaw conducted a research work on studying the effect of an architectural style on the designed system. Her study was carried out using the example of the cruise-control system which was originally presented in [Booch 1986].

In this section, we will briefly introduce a number of common architectural styles. The choice of the selected styles does not imply that they are the best styles for a particular application, but they were selected for their common use in system's organisations and their support to reuse.

3.3.1.1 Pipes and Filters Style

In this style, there are three major elements; *filters, pipes* and *data streams,* these are unique to this style. Components in this style are the filters where each has a set of inputs, and generates a set of outputs by applying a local transformation to the input streams. Pipes are the connectors between the components which are mechanisms for transmitting outputs of one filter to inputs of other filter. Data streams are data flow between components. For each

component there are two streams of data, one at the input and one at the output, sometimes known as *upstream* and *downstream*. Figure 3-5 shows a graphical presentation of this style.

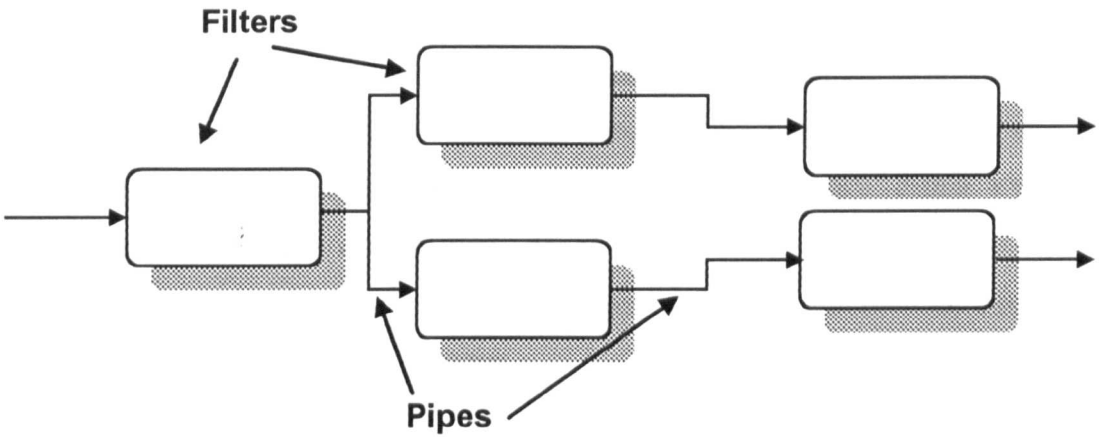

Figure 3-5 Pipes and Filters Architecture Style

This style has the following characteristics:

- Filters must be independent entities: in particular, they should not share state with other filters.
- Filters do not know the identity of their upstream and downstream filters. They might require certain requirements in the input streams or specify the nature of their output streams, but they may not identify the filters which supply or receive the data.
- There should be no constraints (regarding correctness of processing) on the order and organisation of the filters and pipes in a system.

The best known examples of pipe and filter architectures are programs written in the UNIX shell. Another example of this style is the traditional compiler systems.

The following benefits in this style give it wide usage in system architecture:

It allows the designer to understand the overall input/output behaviour of a system as a simple composition of the behaviour of its individual filters.

- It supports reuse: any two filters can be hooked together, provided they agree on the data that is transmitted between them.
- It supports concurrent execution; each filter can be implemented as a separate task and potentially executed in parallel with other filters.

Although literature identifies some potential problems with this architectural style, like possible batch processing organisation, we think that the main disadvantage with this style is its lower degree of flexibility. Once a system is organised using filters and pipes, it is difficult to change the system's configuration to accommodate new requirements. The whole system architecture might need changing for that purpose.

3.3.1.2 Implicit Invocation Style

In implicit invocation, procedures and functions, in a certain component, are not invoked directly by other components. Instead, a component can announce (or broadcast) one or more events. Other components register an interest in a certain event by associating a procedure with the event. When the event occurs the system itself invokes all of the procedures that have been registered for that event. Thus an event occurrence "implicitly" causes the invocation of procedures in other modules.

Components in an implicit invocation style could be thought of as modules which provide procedures (as with abstract data types) and a set of events. So in addition to the possibility of procedures being invoked directly, a component can register some of its procedures with events of the system. The main property of this style is that components that raise events do not know which components will be affected by those events. Therefore, components cannot make assumptions about the order of processing or what processing will occur as a result of their events.

The main benefit of this style is its strong support for reuse. New components can be introduced to the system by registering it for certain events in that

system. Furthermore, components in this style can be modified or replaced by new ones without affecting the interfacing of other components.

The main disadvantage of this style can be summed up by its uncertainty in the performance of the system. The components which announce events do not know what components will be affected by the events and do not know the order of processing and the computation involved which might cause problems especially in safety critical systems. Another problem is associated with data management. Usually data is passed with the event itself, however, sometimes the data is stored in a common repository in the system and managing this data may be problematic in this style.

3.3.1.3 Layered Systems Style

This style is widely known in communication protocols and operating systems. Systems in this style are organised hierarchically in layers where each layer represents a level of abstraction. Each layer provides services to the layer above it and requires services from the one below it. Lower levels define lower levels of interaction, the lowest typically being defined by hardware connections. In some layered systems, inner layers are hidden from other layers except the adjacent outer layers. Some carefully selected functions are excluded and may be exported to other layers or users, such as low level procedure calls in operating systems. Figure 3-6 shows an illustration of this style. Real-time software systems are usually arranged in a similar fashion [Baker and Scallon 1986].

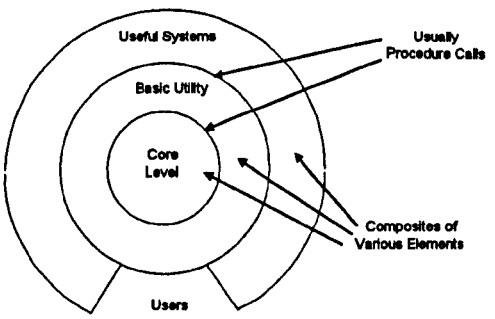

Figure 3-6 The Layered Systems Architecture Style

The layered systems style has the benefit of incremental design by partitioning a system into a sequence of incremental steps based on increasing levels of abstraction. Another benefit is its support to software reuse; different implementations of the same layer can be used interchangeably, provided they support the same interfaces to their adjacent layers. On the other hand, this style suffers from two main disadvantages. First, not all systems can be structured in this way and secondly, due to performance consideration, there may be high coupling between high level functions and their low level implementation.

3.3.1.4 Blackboard Architecture Style

Blackboard systems have originally been used in AI and signal processing applications such as speech and pattern recognition. This architectural style treats problem-solving as an incremental, opportunistic process of assembling a satisfactory configuration of solution elements. [Hayes-Roth 1985].

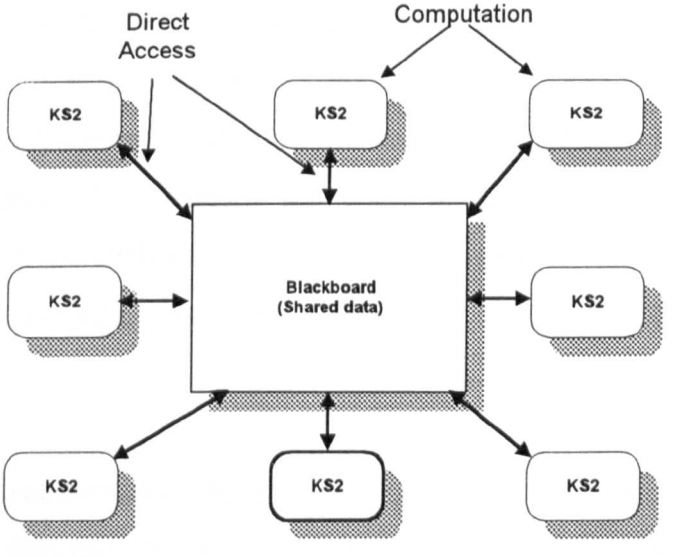

Figure 3-7 The Blackboard Architecture Style

A blackboard system is usually presented with three main elements (Figure 3-7):

· The Knowledge Sources:

Separate, independent parcels of application-dependent knowledge (solution elements), which are generated during the problem-solving. Knowledge sources

have a condition-action format. Only knowledge sources whose conditions are satisfied can perform their actions. Knowledge sources are independent in that they do not invoke one another and ordinarily have no knowledge of each other's expertise, behaviour, or existence.

- The Blackboard Data Structure:

A global database in which solution elements are organised into an application-dependent hierarchy. Knowledge sources make changes to the blackboard that lead to a solution to the problem.

- A Scheduling Mechanism:

Because in most blackboard systems knowledge source activities are event-driven (depending on the condition-action format), they may compete to execute their actions. A scheduling mechanism is needed to determine which activities execute their actions and in what order. The scheduling is driven entirely by state of the blackboard. Knowledge sources respond opportunistically when changes in the blackboard make them applicable.

This architectural style is used to produce a problem-solving style that is characteristically incremental and opportunistic. The knowledge sources generate solution elements, one at a time, and record them in different blackboard locations. They extend the most promising solution elements and eventually merge them with others to form the complete solution. The sequence with which the solution develops depends largely upon the scheduler's behaviour.

The blackboard architecture style supports reuse by providing solutions to domain problems. Domain knowledge sources respond to, generate, and modify solution elements on a domain blackboard, under the control of a scheduling mechanism.

3.3.2 Domain Specific Software Architectures (DSSA)

Currently, there is a growing interest in the study of software architectures for reuse. Many of these efforts have been focused on developing domain-specific software architectures (DSSAs) which are architectures for a family of application systems in a domain [Kogut and Clements 1994]. These architectures are used as a basis for developing systems within that particular domain, thus supporting reuse of design information in the domain.

In July 1991 the Advanced Research Projects Agency (ARPA) launched a program to build a number of domain-specific software architectures known as the ARPA DSSA program. The program comprises developing "reference" architectures for specific domains in a five-year research project [Mettala and Graham 1992]. DSSA is based on the concept of an accepted generic software architecture for the target domain. Some of the domains (mainly military domains) targeted by the project are Avionics Navigation [Cogalianese, et al 1992], Guidance and Flight Director [Agrwala, et al. 1992], Command and Control [Braun, et al. 1992], Distributed Intelligent Control and Management (DICAM) for Vehicle Management and Intelligent Guidance, Navigation and Control [Hayes-Roth, et al. 1992].

The main question asked is; what is a DSSA? Will Tracz [Tracz 1994] defines the DSSA as:

> ... a process and infrastructure that supports the development of a Domain Model, Reference Requirements, and Reference Architecture for a family of applications within a particular problem domain. The expressed goal of a DSSA is to support the generation of applications within a particular domain.

From this definition we could identify the elements of a DSSA which are

- A *software architecture* (sometimes called reference architecture) with reference requirements and domain model.

- *Infrastructure* to support it.
- *process* to instantiate/ refine it.

Figure 3-8 [Tracz 1995] shows a typical DSSA reference architecture and infrastructure.

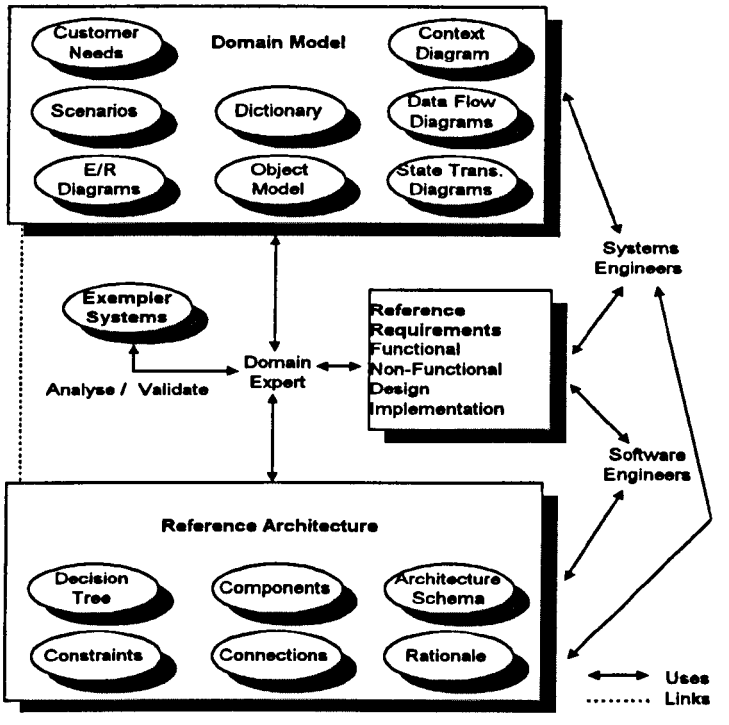

Figure 3-8 DSSA Artefacts

The difference between reference architecture and application architecture is that, an application architecture is an architecture for a single system and a reference architecture is a software architecture for a family of application systems. Reference architectures support reuse by providing a design model for all systems in the problem domain, and normally are refined to generate an application architecture. The infrastructure is a collection of reusable components that resides in the domain model ready for re-application. These components could be code fragments, domain dictionary, scenarios, object model ..etc. (see Figure 3-8). Reference requirements are behavioural requirements for applications in a domain used to drive the design of the reference architecture.

3.3.3 Review of Research in Software Architectures.

In this section, we will review some examples of reference architectures, for specific application domains. The first effort was presented (as stated earlier in

this chapter) by the ARPA DSSA program [Mettala and Graham 1992], which is a project for building reference architectures for a number of military domains. As an example of this program the DICAM (Distributed Intelligent Control and Management) project is reviewed first.

The DICAM-DSSA project is developed simultaneously as a "model" or framework for understanding control problems and as an architecture and related environments for the rapid development of high performance controllers to be employed in DICAM applications [Terry, et al. 1994]. In the process of building controllers, concepts from software engineering and knowledge engineering are combined in a software development environment. This environment includes a blackboard-like development workspace to represent both the software under development and the software development process.

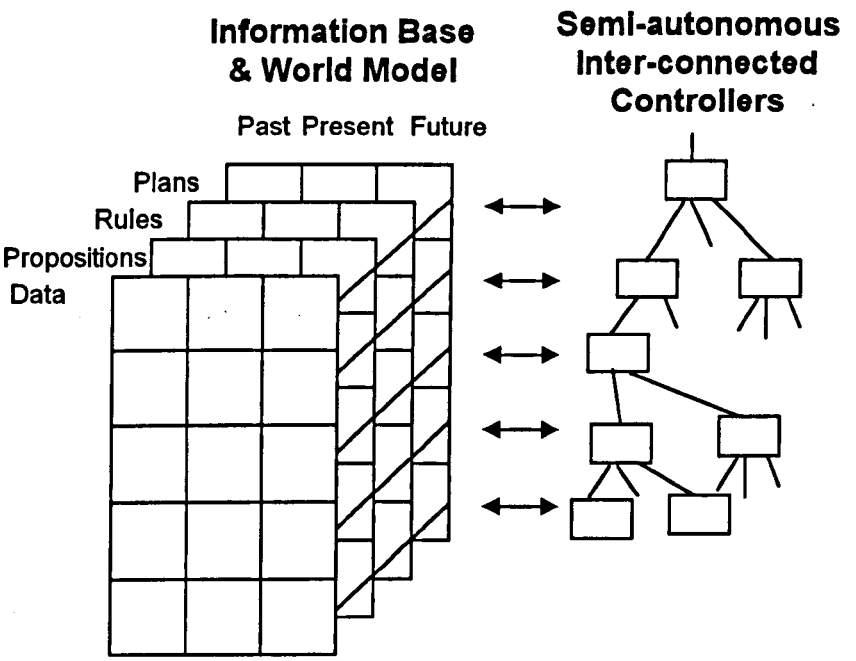

Figure 3-9 The DICAM Vehicle Architecture

DICAM-DSSA uses a reference architecture for modelling the interaction of controllers in a DICAM application, and internal structure of an individual controller in the reference architecture. These controllers may work as a single intelligent agent or as a multiple co-operating agents. Figure 3-9 [Terry, et al. 1994] illustrates the DICAM reference architecture. This architecture provides a

general model of controller applications that prescribes the key system components and their inter-relationships. It includes two principal components in any distributed intelligent control and management application. First an information base and world model is a conceptually distributed in a centralised database/ knowledge base that represents the state of the world. It can be viewed as a three-dimensional structure. The first dimension represents information stored at high to low-levels of aggregation of controllers' corresponding levels of responsibility. The second dimension corresponds to the different types of information that must be stored. Four types of information are shown on the figure termed data, propositions, rules and plans. The third dimension is time.

The second principal component of the DICAM reference architecture is a collection of semi-autonomous interconnected controllers. The controllers are differentiated in terms of the scope of behaviour they address, the resources they control and the time frame spanned by their decisions.

In [Hayes-Roth, et al. 1995], a domain-specific software architecture for a large application domain of adaptive intelligent systems (AIS) is presented. This DSSA has three main elements; first it provides an AIS reference architecture designed to meet the functional requirements shared by applications of the domain. Secondly it provides principles for decomposing expertise into highly reusable components and the third element is a configuration method for selecting relevant components from a library and automatically configuring instances of these components in an instance of the architecture.

The AIS reference architecture is a heterogeneous mixture of common architectural styles (Figure 3- 10 [Hayes-Roth, et al. 1995]). It is divided hierarchically into layers for different sets of computational tasks. The layers and the relationships among them provide properties of pipe and filter style architectures. Each layer, itself, comprises a number of components, organised in a blackboard style, to allow for a range of potentially complex behaviour. The architecture has two layers, or levels, to control concurrent physical and

cognitive behaviours. Behaviours at the physical level implement perception and action in the external environment. Behaviours at the cognitive level implement more abstract reasoning activities such as planning, problem solving, etc. Information flow is bi-directional. The results of cognitive behaviours can influence physical behaviours and vice versa.

As shown in Figure 3- **10** this AIS reference architecture has some features which are inherited from the DICAM reference architecture. Firstly, the blackboard structure is implemented in the Information Base and World Model which has the same three dimensional view. Secondly the notion of meta-controllers is used in both architectures.

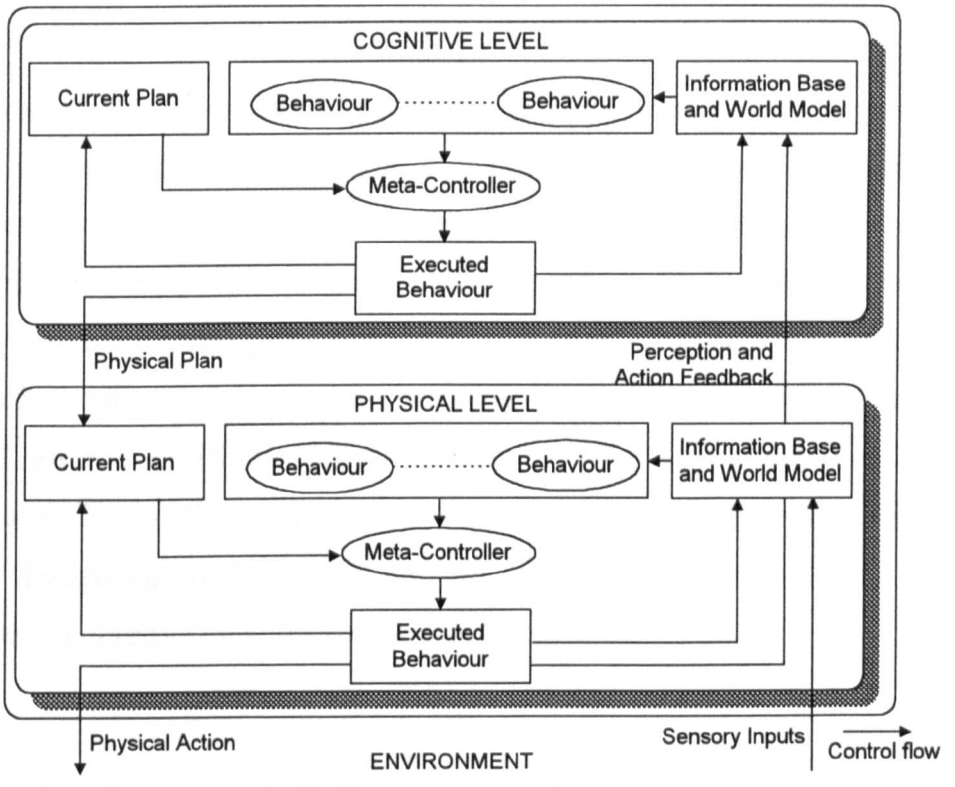

Figure 3- 10 AIS Reference Architecture

There are some other research efforts for building reference architectures for specific domains. These are not domain specific software architecture (DSSA's include, in addition to reference architectures, an infrastructure and a process).

In [Baker and Scallon 1986], an architecture for real-time systems is proposed. The architecture is based on the layered style, where layers are arranged from low to high-levels of abstraction. At the low-level layer is the hardware which is followed by the operating system layer that handles interrupts, manages hardware configuration and detects faults. The operating system interacts with the next layer (the executive) through a standard interface. The executive schedules application processes, allocates storage and dispatches tasks within processes. The highest level in the architecture is the application layer which also uses a standard interface to interact with the executive. The application layer performs tasks for solving user's problems and is independent of hardware details. The paper also proposes an architecture for the executive layer by using three components for performing the executive's tasks. These components are, *scheduler*, *resource allocator* and *dispatcher*.

In another effort [Shaw 1995], a new approach for the design of process control software was proposed based on the control architecture. This paper uses the original control view of the feedback control problem to solve object-oriented design problems for process control software. The author used the cruise control problem, which was first presented by Grady Booch in his paper [Booch 1986], to illustrate how this view is used for building an architecture for the cruise control system. Figure 3-11 [Shaw 1995] shows the proposed cruise control software architecture. The author argues that this problem is a control problem and the software design could be based on the control view architecture which, she claims, would clarify the software design.

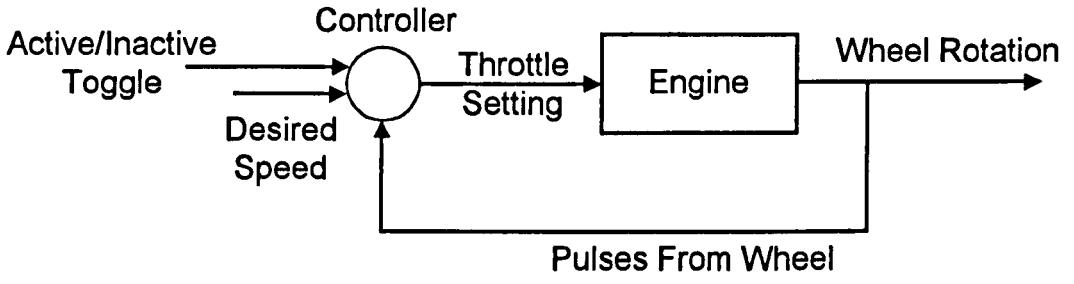

Figure 3-11 Control Architecture for Cruise Control

We cannot consider this view as a reference architecture for the process control domain (regardless of its appropriateness) for the following reasons:

1. There are a number of views to the control problems (e.g. feedback, feed forward, adaptive control, etc.) which need more than one architecture to model and this might confuse the user of the reference architecture.
2. This view is a design tool (the block diagram method) used by control engineers to analyse physical systems and derive their transfer functions (mathematical model of the system's physical behaviour). In many cases the block diagram method is a simplified version of the actual control organisation. On the other hand, the software components may be arranged in a different order to satisfy constraints imposed by the operating system, memory mapping or performance.
3. There are situations in the process control domain that cannot be represented using this control view. For instance, batch processes which are implemented as a sequence of actions cannot be modelled using this 'block diagram' approach'.

In [Kazman et al. 1994], a method for analysing the properties of software architectures is proposed. The paper describes three perspectives for understanding the description of a software architecture and then proposes a five-step method for analysing software architectures called SAAM (Software Architecture Analysis Method). These analysis perspectives are the functional partitioning of its domain of interest (the application), its structure and the allocation of domain function to that structure. The authors illustrate their method by analysing three non-commercial user interface architectures with respect to the quality of modifiability.

3.4 Summary

To summarise the points covered in this chapter, we start by differentiating between design patterns and software architectures and then explain how each supports software reuse. An application architecture represents a model of the organisation of a single software system and a reference architecture represents

an organisation of common components and the inter-relationships among them for a family of systems in a specific application domain. A design pattern is a possible arrangement of classes (or objects) and their structures and interfaces which provides a solution to a design problem that is found across a wide range of object-oriented software systems. A design problem could be a generic one (found in any domain), or specific to certain domain (e.g. specific to user interface domain).

Software architectures support reuse through the use of reference architectures that can be refined to generate an instance of the application architecture for the system under development. On the other hand design patterns support reuse through encapsulating design expertise in the pattern which can be applied for solving that particular design problem over and over again.

Reference architecture is a term which means a software architecture for a family of systems that could be instantiated into system's architecture when systems are developed. The domain specific software architecture is a term used by ARPA to denote a program for developing reference architectures for a number of related domains. The discussion of domain specific architectures and reference architectures showed that architectures for similar domains usually employ a common style. For example, all reference architectures for real-time domains are based on the layered style and reference architectures for intelligent domains use the blackboard architectural style. There is no approach to define a generic architecture model (or models) that can be used or re-organised to build reference architectures for specific domains. Such type of architectures could prove to be very useful in addressing design and organisation issues across a wide range of domains.

In the next chapters, a new approach for building reference architectures is described. The approach defines a number of generic models that are tailored according to domain-specific constraints to build a reference architecture. The generic software architectures is a new term used to describe a new technology

for building reference architectures from pre-specified architectural models. The reference architecture, which is an architecture for a family of systems within the domain, is then used for defining the interaction between components when systems are synthesised. Reference architectures comprise a number of architecture schemas which represent design conceptions in terms of reusable components and relationships among them. Architecture schemas are different from design patterns in the sense that design patterns represent solutions to typical object-oriented design patterns. Whereas the architecture schemas encapsulate domain-specific design decisions that are used when systems are built from reusable components.

Chapter Four

4. DOOR - An Approach to Domain Oriented Object Reuse

4.1 Introduction

In this chapter, we discuss a new approach to software development based on reusable objects. As opposed to generic components, we propose domain-oriented components that are specifically designed to be re-applied in a specific scope within the application domain; thus the name Domain-Oriented Object Reuse (DOOR). The approach combines development for reuse and development with reuse together and is based on the existence of a domain knowledge-base which is also referred to as a domain model. The approach is divided into two phases, domain engineering and application engineering, which are performed in parallel throughout the development process. In domain engineering, the domain model is built and its components are assessed, whereas in application engineering, the domain model is used as a framework for synthesising new systems.

Our approach is based on the following elements:

1. A *domain infrastructure*, called *DOOR Assets*, that support the development of new systems. The infrastructure comprises a domain taxonomy, reusable components and a reference architecture.
2. A new *technology* called *Generic Software Architectures* which is used in developing and refining the infrastructure. This is encapsulated in a number of *Generic Architectural Models* which are used for describing the reference architecture.

3. A *process* and *guidelines* for building the infrastructure (using the technology) and for synthesising new systems using DOOR Assets. The process is divided into *Domain Engineering* and *Application Engineering.*
4. A supporting tool for automating and validating the infrastructure development process; see chapter seven and figure 7-2.

In this chapter, we will provide an overview of the approach and a description of the infrastructure which is known as DOOR Assets. In the next chapter, the technology of Generic Architectures is described and in chapter six, we will describe the process. The tool support and a case study are reported in chapter seven.

4.2 Problem Statement and An Overview of the DOOR Approach

Two main problems have been identified with domain analysis methods which are: first, most methods are conducted on ad-hoc basis where no guidelines are specified to produce the products of domain analysis; secondly, there is no systematic approach for retrieving and applying these products in the development of new systems. Other problems with reusing components are related to how a software component is specified such that its functionality is modelled. Furthermore, how could we locate and retrieve components that meet the requirements of the new system. This approach suggests possible solutions for these problems. It provides a number of guidelines to model domains and classify them in a hierarchical classification of domain abstractions, known as the domain taxonomy, from which components are traced and retrieved by narrowing the search space to a specific abstraction level, known as the scope of reuse. Components are specified using a 3-D model, which specifies the component's scope, behaviour and relationships with other components in the domain model (see section 4.4.2). The relationships with other components are specified using the generic software architectures technology which helps in locating the reusable components.

In our approach, which is based on domain analysis, software systems are developed from reusable components, objects and their specifications, that are especially developed to be reused within an application domain. The DOOR approach re-configures the software life-cycle for designing domain oriented components by using a domain knowledge-base, called a *domain model*, to assist in the identification and specification of reusable components [Al-Yasiri and Ramachandran 1994]. The domain model contains the domain infrastructure (DOOR Assets) which can be incorporated while synthesising new systems. Every time a new system is constructed, the DOOR assets and the reuse effort are assessed, and where applicable the domain model is updated. Figure 4-1 shows the main elements of the DOOR approach.

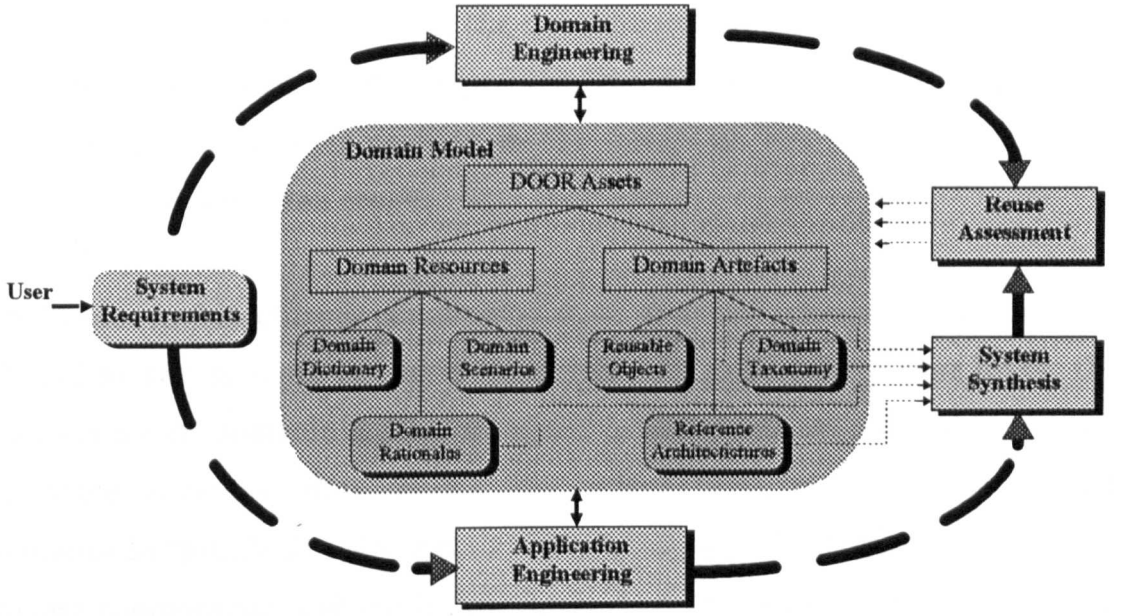

Figure 4-1 Elements of the DOOR Approach

DOOR assets are divided into *domain resources* and *domain artefacts*. The distinction between the two types is attributed to the way the assets are prepared; the resources are collected and presented in the domain model, whereas the artefacts are developed by the domain engineer and added to the domain model. The resources are information items that are collected during the domain analysis process and comprise dictionaries, scenarios and rationales. The purpose of the resources in the domain model is to capture the domain-specific

information and present them to the domain engineers and systems engineers. The artefacts are the reusable assets that are generated during domain engineering and comprise a domain taxonomy, reusable components and the reference architecture. The domain taxonomy classifies the domain according to the scope of application of its reusable components. It is represented as a tree of the domain's subject matters and their corresponding components. The taxonomy is useful to introduce the domain organisation to the users where they can focus their search for reusable assets to a specific scope on the tree.

A reusable component in DOOR means, a packaged piece of code (encapsulated in an object) and its specifications. This term may be used in a different context in other approaches to mean any piece of information (including code, design, frameworks, etc.) that are used in the development of a number of applications. We use the term DOOR asset to represent this notion. Reusable components are specified according to their position in the taxonomy tree to perform a specific task within its application scope.

The reference architecture is constructed from generic architectural models, each of which has two or more reusable components, a relationship that links them and a number of constraints. The reference architecture consists of a number of architecture schemas that model the interactions between the reusable components to specify certain domain-specific design conceptions. It also helps in tracing components and retrieving them from the domain model. A detailed description of the generic software architectures is found in the next chapter.

The process of building the domain model is integrated in the domain engineering and application engineering phases. The process is supported by a sets of guideline for building the taxonomy, the reference architecture, synthesising systems and assessment of the reuse effort. The detailed description of the process is presented in chapter six.

4.3 The Domain Resources

During domain analysis, the whole application domain is analysed. This is mainly a knowledge acquisition process for gathering as much information as possible about the domain and building the domain model. The results of the domain analysis are called *Domain Resources*, which are information items collected from the domain sources of information (for details see chapter two) and stored in the domain model as text for future reference. In DOOR, domain resources comprise *Domain Scenarios, Domain Rationales* and the *Domain Dictionary*. Domain scenarios represent domain specific transactions, domain rationales represent domain specific constraints or non-functional requirements whereas a domain dictionary is a description of domain specific terms and concepts. In domain analysis we try to model the problem space of the domain, therefore the domain resources reflect the state of the problem space not the solution space. However, these resources are used for eliciting artefacts in the solution space.

In general, domain resources do not require further refinement apart from sorting and combining together related resources. The resources are cross referenced between each other. For example, a certain domain scenario could be linked to one or more rationales which are relevant to the scenario, or the domain dictionary may contain information that explains a certain concept that occurs in a certain scenario.

4.3.1 Domain Scenarios

These are extracts from functional requirements of existing systems in the analysed domain which are common in the application domain. They are vital for understanding usual transactions in the domain. An example of such a scenario is shown in Figure 4-2 . They are also useful for identifying objects in the domain using textual analysis methods; where objects correspond to nouns and methods to verbs. We must emphasise at this point that the objects identified from the scenarios may well be subject to change at later stages of the analysis process as the process is iterative.

Pump Control Scenario

The required behaviour of the pump is that it monitors the water level in the sump. When the water reaches a high level (or when requested by the operator) the pump is turned on and the sump is drained until the water reaches the low level. At this point (or when requested by the operator) the pump is turned off. The pump should only be operated if the methane level is below a certain critical level.

Figure 4-2 An Example of a Domain Scenario

When the domain model is built the scenarios play a major role in identifying reusable components and sub-systems in the domain. By examining the previous scenario, we could identify some useful objects for pump control systems. Examples are: pump, sump, sensor and operator. Some of these objects could be generalised as domain-oriented components and added to the domain model. In addition the scenarios could also be used for defining dependencies among objects, which is the basis for specifying reference architectures in the domain model.

When new systems are synthesised the domain scenarios are used for matching the system requirements with the domain assets. They are then used for providing reusable specifications of the system components. By identifying the relevant scenarios the reuser will be able to identify a number of components from the domain infrastructure whose specifications are already included with them.

4.3.2 Domain Rationales

Domain rationales are closely related to the *scope* of reuse within the domain. A reuse scope is defined as an abstraction level in the domain structure where a component or group of components are likely to be applied; more discussion about the reuse scope is found in section 4.4.1. For every scope in the domain,

there are a number of rationales that are associated with it. Despite the fact that these rationales represent constraints that restrict the design of the components, they are essential for ensuring that components comply with the domain specific requirements and hence increasing the chance of reusing them. Normally, domain rationales are non-functional requirements, design and implementation requirements extracted from existing systems within the domain. Such requirements serve the development of systems or sub-systems in the domain by providing a general view for consideration. These are limiting characteristics in the solution space of the domain artefacts. This could be in the form of time, space or specific language or platform constraints. An example of such rationales is shown in Figure 4-3.

Response Time Requirement:
For all interrupt service routines, the maximum allowed computation time should be in the range of **1 ms**. This is a re-assurance for regular pulse rates generated from different timers in the system.

Figure 4-3 An Example of Domain Rationale

4.3.3 Domain Dictionary

The third element of the domain resources is the domain dictionary. This is a list of commonly used words and phrases found in the scenarios and customer needs documents. The dictionary includes a brief description about every word or phrase in the list. The dictionary acts as a supporting information tool to other resources. However it is an important part of the domain resources and should be constructed carefully and accurately. It should give concise but precise description about its elements and domain experts must be consulted when it is constructed.

As with other resources, the domain dictionary is organised according to the reuse scope. This reduces the effort of looking up entries in the dictionary and makes it easier to link it with other resources.

4.4 The Domain Artefacts

The domain artefacts are the elements of the domain infrastructure that are generated by the domain engineer using the domain resources. The artefacts are not collected from the domain information sources (as in the domain resources), but they have to be produced and modelled during the domain engineering process by following a number of steps as will be described in chapter six. The domain artefacts include *Domain Taxonomy, Reference Architectures* and *Reusable Components.*

4.4.1 Domain Taxonomies

An application domain is defined as a class of similar systems which provide services to that application and share common features and objects. Each system in the domain can then be called a member system of that domain. If a domain is modelled, the common features in the domain are identified and their reuse scope is specified. However, it is found that analysing the uncommon features in the domain is also necessary to complete the picture of the domain model.

The domain member systems may share some components in terms of objects and functions. Such components will then have reuse scope across the entire domain. Some of the systems may be grouped in subsets within the main domain. Each subset will have common components shared by them. Each subset is then called a sub-domain and could be decomposed in the same way as the main domain. Sub-domains represent abstraction levels and usually outline subject matters within the main domain. For example the domain of networking software could be decomposed into Local Area Networks (LAN) and Wide Area Networks (WAN) sub-domains.

The common components in the domain are grouped together and included in the taxonomy tree. They are indicated in the core of the taxonomy tree and called the *Domain Kernel.* The kernel contains frequently reusable components in the domain whose scope covers the entire domain. These components are identified and specified using object-oriented analysis methods [Graham 1991].

The remainder of the domain member systems is called the *Domain Subordinate* as shown in Figure 4-4. The domain subordinate contains specifications of the domain member systems less the common components which are identified in the domain kernel. Systems in the subordinate inherit the features of the kernel components of the main domain. We, therefore, refer to the main domain as the parent domain of the subordinate.

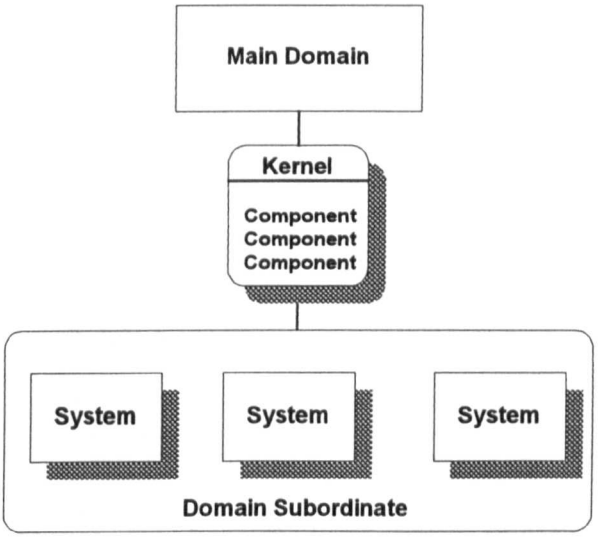

Figure 4-4 The Domain Structure

More reusable components in the subordinate may be identified, but their reuse scope will not cover the entire domain. They could have a scope for reuse within a subset of the application domain. This is why modelling the uncommon features is necessary in this approach. By identifying these subsets, the domain subordinate is then divided into a number of sub-domains which can be modelled in the same way. The process continues to structure the domain into sub-domains and kernels of reusable components until no further decomposition is possible. Such a case is identified on the tree as a *Unity Domain* which is a domain abstraction whose functional requirements can be conceived by one system, and perceived by the domain community as a common utility in the domain.

The domain taxonomy is represented graphically (as shown in Figure 4-5) using the following elements:

1. A Main domain (denoted by rectangular box),
2. Sub-domains (denoted by rectangular boxes),
3. Unity domains (denoted by double-lined boxes),
4. Kernels (denoted by rounded boxes), and
5. Reusable components which are listed in the kernel

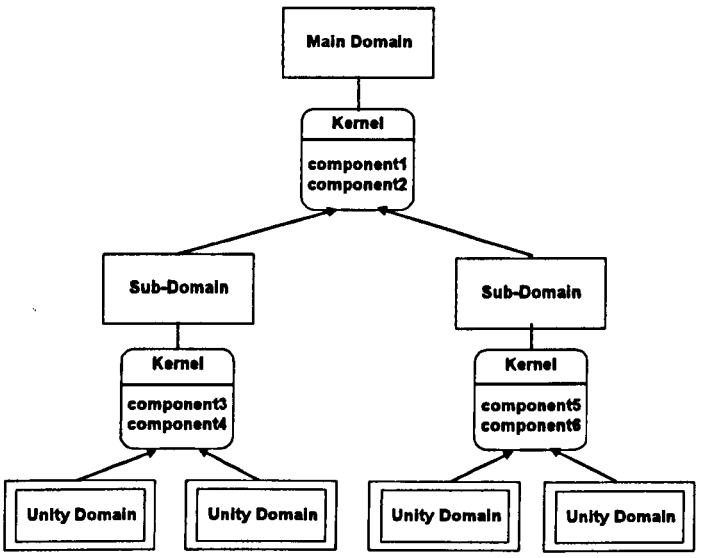

Figure 4-5 Domain Taxonomy Tree

There are several benefits gained from classifying domains and building a structural model of the domain classification. We can summarise these benefits in the following points:

1. The main aim of conducting domain analysis is to capture the domain knowledge and present it to developers who intend to make use of the reusable assets in the domain model. The structural model presents this knowledge in a more understandable and easier to follow fashion.
2. The structural modelling of domain knowledge focuses the search for the relevant information to the relevant application domain. This minimises the effort spent in tracing the domain assets and identifying components.
3. The domain structure clearly identifies the reusable components in the model. In fact, as soon as the scope of application is selected, a list of reusable components can be obtained from the domain kernel. From these

ones, other components could be traced, by following the inter-relationship links between them. These links are also included in the domain model (see chapter five).

4.4.2 Reusable Components.

As we have said in the last section, reusable components in DOOR are represented by objects and their specifications. Each component is designated a scope for reuse. This means that any reusable component must be associated with one sub-domain and included in its kernel on the tree. This also means that a reusable component must comply with the constraints imposed by the domain scope in the way the component is designed or the way it interacts with other components. Therefore the component location on the taxonomy tree is important for determining its scope and constraints.

In DOOR, a reusable component is classified according to its scope or its location on the taxonomy tree as:

- *Intrinsic,*
- *Frequently Reusable,*
- *Candidate* or
- *Bounded.*

For a specific scope in the domain taxonomy, components that are inherited from the parent domain kernel are called *Intrinsic* components of that scope. Any component that is part of the kernel of that level is called *Frequently Reusable* component within that scope. Components that are located in the kernels of the subsequent levels are called *Candidate* components. If the subordinate of that level contains any unity domains then these are called *Bounded* components. Figure 4-6 shows a graphical representation of the components' classification according to scope. Any components that are located outside the scope of that level are said to have no scope for reuse within that level. For example, the components *Comp 3* and *Comp 4* in Figure 4-6 have no scope for reuse within *Domain 1.* For examples of component types according to scope, see section 4.5 where the domain of reservation and inventory systems is classified. It must be

noted that at a main domain level, the intrinsic components and the frequently reusable components are the same; this is because a main domain has no parent domain scope.

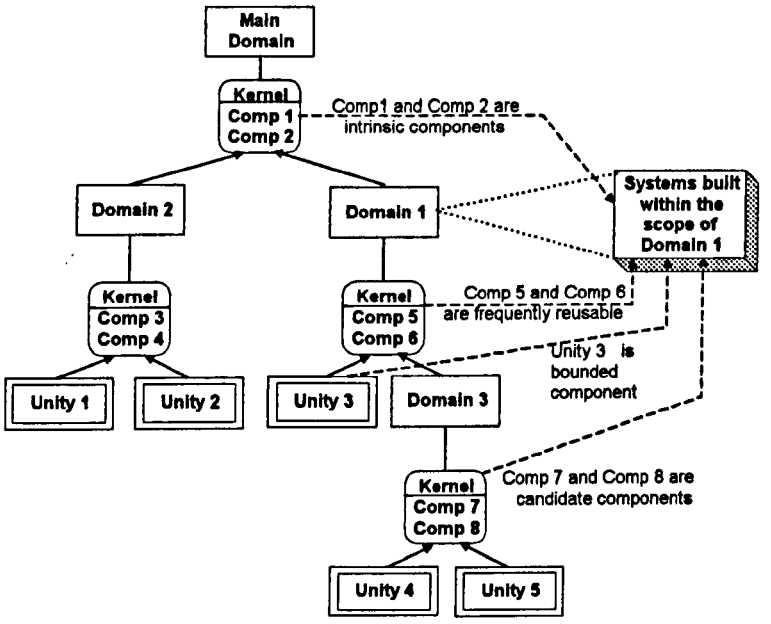

Figure 4-6 Classifying Reusable Components according to scope

A reusable component is modelled using what we call the 3-D model as shown in Figure 4-7. This model helps understanding the semantics of the reusable components. It models a component in terms of its *behaviour, scope* and *reaction.*

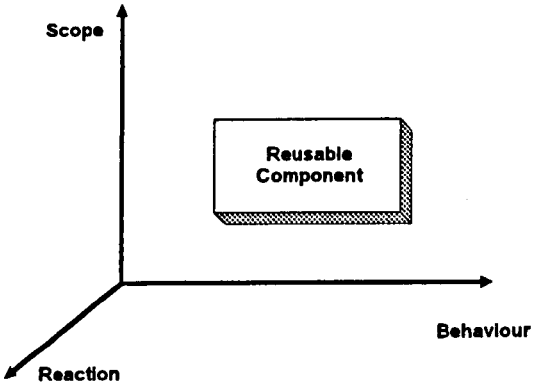

Figure 4-7 3-D Model of the Component

The first dimension is the component's behaviour which is characterised by the component interface; signature of all methods of the object. In DOOR, a reusable component is stored in the domain model with its interface methods and specifications. The component's behaviour is important for building the reference architectures as will be discussed in the next chapter. The second dimension of the model is the domain scope. It represents the position of the component on the taxonomy tree. The scope imposes a number of constraints which must be complied with by the component behaviour. The third dimension is the reaction of the component which represents all the components that are linked with it in the reference architecture. It is important to note that the reaction space of any component, which are the components that are allowed to be liked to that component, is determined by the four component categories of its scope; intrinsic, frequently reusable, candidates and bounded components. In this case the reaction space of any bounded component is its intrinsic components only.

4.4.3 Reference Architectures.

Reference architectures are the links that model the relationships between all other artefacts in the domain model. They are built using the reusable components and the interrelationships between them. The way the components are interrelated is governed by the domain constraints. These constraints are embedded in a number of generic software architectures and a domain description language.

Reference architectures depend on the reusable components, their behaviours and their scope. They are built using architecture schemas which are constructed from components that appear in each others' reaction space. The detailed description of the technology of generic software architectures and the way reference architectures are built will be discussed in the next chapter.

4.4.4 Guidelines for Building Domain Taxonomies

Structuring the application domain into a tree of abstraction levels is important in the DOOR approach because the domain resources as well as the domain

artefacts are designated to the relevant scope in the tree. The domain tree is also useful in tracing and retrieving the domain assets from the domain model. Hence it is important to have a tree that reflects the real structure of the domain and helps in tracing the reusable assets in the domain model.

In this section we introduce some guidelines for structuring the domain taxonomy tree. These guidelines could be conducted according to the order presented here or arbitrarily as regarded necessary by the domain analyst. We must also say that it is likely that several attempts are conducted before a satisfactory tree is achieved. On the other hand, as a rule of thumb, the domain taxonomy is constructed from the application subject matters and their systems. It is also important to say that this approach looks like a top-down approach, however it is possible to apply a bottom-up approach to build it by identifying components and systems first then group them into sub-domains.

Guideline 1

Apply domain analysis by considering existing systems in the domain and with the aid of a domain expert. Prepare a list of possible reusable components within the domain and extract domain scenarios by analysing existing systems and interviewing domain specialists.

Guideline 2

Run through the domain scenarios and identify a list of possible operations in the domain that can be automated and group them according to behaviour abstractions and designate a name for each behaviour which is perceived by the domain community as common utilities (or systems) in the domain.

Guideline 3

Define a number of subject matters within the domain and allocate them to groups of the systems identified in guideline 2; these correspond to sub-domains in the taxonomy. As a general rule each sub-domain must have at least three systems allocated to.

Guideline 4

Identify reusable components within the sub-domain and group them together as kernels for that sub-domain.

Guideline 5

Identify related components (objects) that are allocated to more than one sub-domain in the same level and design an abstract class which includes only the common features of all the identified components. Such abstract class should be included in the kernel of the parent domain of that level.

Guideline 6

If one ore more abstract classes are identified for a subset of the subordinate then a new level in the tree could be introduced which will have a kernel of abstract classes only. The same rule of at least three systems in one sub-domain must be maintained in this case as well.

Guideline 7

Any systems that are left ungrouped with other systems should be identified and modelled as unity domains on the tree.

Guideline 8

In a case where there is a subject matter in the domain whose member systems do not have common components then it is allowed to be identified as a sub-domain (with no kernel) and included it on the tree only if the following conditions are satisfied:

1. *The modelling of this sub-domain is important for understanding the domain model and its resources.*
2. *The subordinate of this sub-domain contains two or more sub-domains and each one of them has its own kernel.*

Guideline 9

Make sure that all the lowest levels in any branch of the tree contain only unity domains.

Guideline 10

Re-visit the whole tree as many times as needed until no further decomposition or grouping among its components is possible.

The above guidelines are general guidelines for identifying domain abstractions and components within an application domain. These are based on our experience that we have gained by applying this approach to some example domains (see chapter seven). Other decisions made during the building of the reference architectures could influence the way the taxonomy tree is constructed. We will introduce more design guidelines for building the reference architecture in the next chapter; some of them are also relevant for the purpose of building the domain taxonomy and its components. Also during the processes of domain engineering and application engineering the domain taxonomy may evolve further. More design guidelines within the DOOR process are also applicable to the construction of the domain taxonomy and the identification of reusable components. Two more sets of design guidelines are found in chapter six.

4.5 Example - Reservation Systems Domain

This example introduces a working case study based on a paper written by Will Tracz [Tracz 1995]. We have chosen this example for the following reasons:

1. The example is already published and presented on more than one occasion which makes it well understood.
2. The example describes a well known domain that many people are familiar with.
3. Although the information about the analysed domain in the paper are concise, it still gives the reader a reference for checking the flow of information and makes it possible to follow the concepts.
4. The example is presented using the DSSA approach and this enables us to compare the results of our approach to the DSSA approach.
5. The example introduces a small and simplified domain which makes it suitable for introducing the concepts of the DOOR approach. A more lengthy example for analysing a real-world domain is described in chapter seven.

This is a working example which will be used in the next two chapters as well. The overall modelling of the domain is built in stages as the approach phases are introduced. In this chapter we introduce the problem and build a preliminary domain taxonomy. In the next two chapters, the same example is used to identify components, relationships between them and the reference architecture is then constructed. You will find out that the solution is maturing gradually as more information is analysed and the final version is achieved after several iterations.

Tracz's Problem Space.

Tracz's paper introduces a simple system for theatre reservation system which could be generalised to cover similar situations in other domains using DSSA's. The information presented in the paper (see also Appendix A) gives a solution to that particular system from the theatre domain and then finds similarities with three more domains, Airline, Library and Inventory domains.

There are a number of scenarios, functional and non-functional requirements drawn from the theatre example which act as resources for driving most of the solution steps in the example. Five scenarios have been included in the paper describing some common transactions in the theatre domain which are Ticket Purchase, Ticket Return, Ticket Exchange, Ticket Sales Analysis and Theatre Configuration Scenarios. These scenarios are found in Appendix A.

The paper first identifies a number of objects in the theatre domain and then generalises them over the other domain. Some of the objects are introduced with a list of attributes and operations. The paper also describes some structures between objects in terms of inheritance and aggregation diagrams. Data flow diagrams and state transition diagrams are used for showing the flow of information and control in the developed system in the theatre domain. Some of the results in the paper are presented in Table 4-1 and Table 4-2. The overall DSSA solution is found in Appendix A.

Table 4-1 List of Objects, Operations and Attributes in the Theatre Domain

Object	Attributes	Operations
Seat	Name Status (e.g. Sold, Available)	Sell a Seat Return a Seat Initialise a Seat
Row	Name	No. of Available Seats List Available Seats List all Seats Initialise a Row
Section	Name (e.g. orchestra)	List All Rows List Available Rows Initialise a Section
Theatre	Name Total Tickets Sold Total Tickets Unsold Total Sales	List Sections Display Seating Arrangement Initialise Theatre

Table 4-2 Comparison between Theatre, Airline, Inventory and Library Domains

Theatre Domain	Airline Domain	Library Domain	Inventory Domain
Seat	Seat	Book	Item
Row	Row	Shelf	Room/ Shelf/ Bin
Section	Ticket Category	Section	Aisle or Building
Performance	Flight Number	Title	Description
Seating Arrangement	Seating Arrangement	Floor Plan	Warehouse
Tickets Sold	Tickets Sold	Books on loan	Items Sold
Tickets Remaining	Tickets Remaining	Books available	Current inventory
Price	Price	Penalty for lateness	Cost/ Item
Performance Time	Flight Departure	N/A	N/A
Performance Date	Flight Date	Due Date	Expiration Date
Ticket Agent	Ticket Agent	Librarian	Clerk

The results shown in Table 4-1 and Table 4-2 are achieved by applying guidelines 1 and 3 of the guidelines for building domain taxonomies introduced in section 4.4.4. By applying guideline 4 and analysing Table 4-2, we could easily find some common components between all four subject matters. As a matter of fact, from experience with building systems of that nature all four situations share more common features than just common objects, such as the reservation or ordering seats or items and looking up a certain item in a list to check if it is available or not. Therefore it is logical to treat them as four sub-domains of one main domain which will have a kernel containing the common objects and each sub-domain will contain its own kernel of objects for that scope. The information in Table 4-2 tells us about the similarity, however it does not tell us where we could reuse the information over the three sub-domains. For example, we know that there is similarity between *seat* object or a *book* object, but we obviously

cannot use them interchangeably. The user would wonder how reuse could be achieved. It is more logical and time saving to identify similarities and suggest ways of reusing them at the same time by showing the relationships between them. In this case a book, a seat and an item could be treated as special cases of one general case which we refer to as *Unit*. Unit will then be an object that contains the common features of all the other objects and could be inherited by them (guideline 5). This represents a case for identifying relationships between components in different scopes. More discussion about components' relationships is found in the next chapter. Similarly we could identify other components that could be used in all four sub-domains. By applying guidelines 2 and 7, we identify some unity domains in the theatre sub-domain. According to the available information, unity domains that are perceived as common transactions within the theatre application are Ticket Purchase, Ticket Return, Ticket Exchange, Ticket Sales Analysis and Theatre Configuration. A preliminary classification of the domain is presented in Figure 4-8 where we may identify some unity domains for the other sub-domains. As an example, we have identified cases for the library domain as unity domains similar to those of the theatre domain.

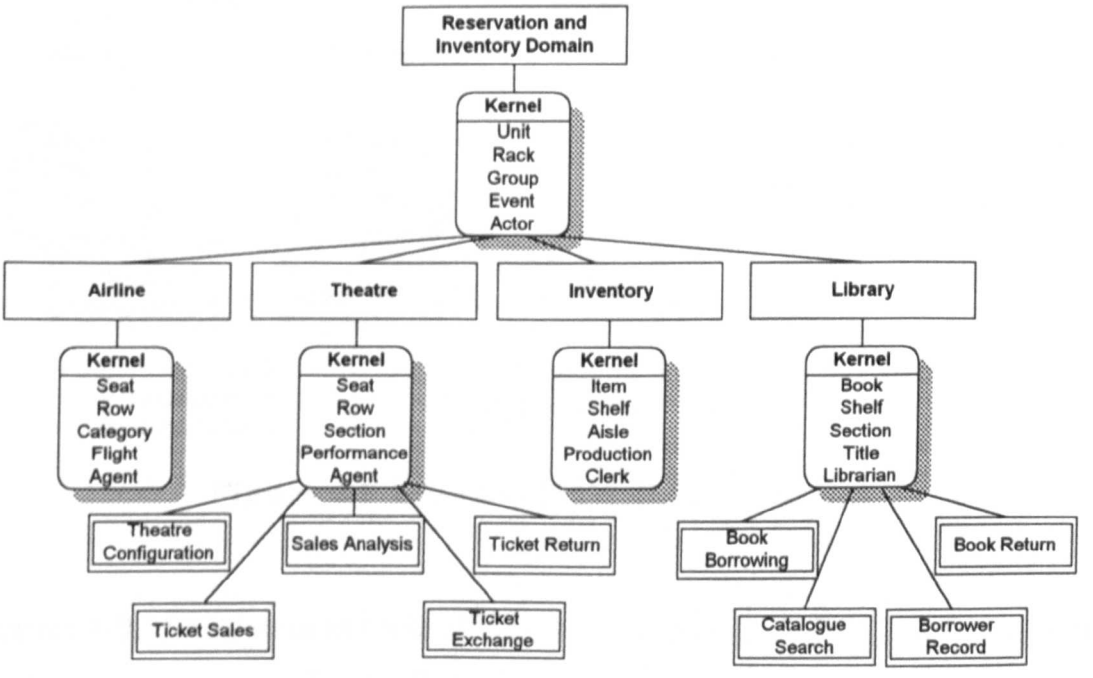

Figure 4-8 Reservation and Inventory Domain Preliminary Taxonomy

By examining Figure 4-8 and applying guidelines 5 and 6 further similarities could be identified between the theatre domain and the airline domain in terms of shared objects. This leads to modifying the taxonomy to reflect this situation by introducing a new level in the taxonomy; a reservation sub-domain whose subordinate includes the airline and theatre sub-domains and its kernel will contain the shared objects between the two. Figure 4-9 shows the modified taxonomy of the overall domain.

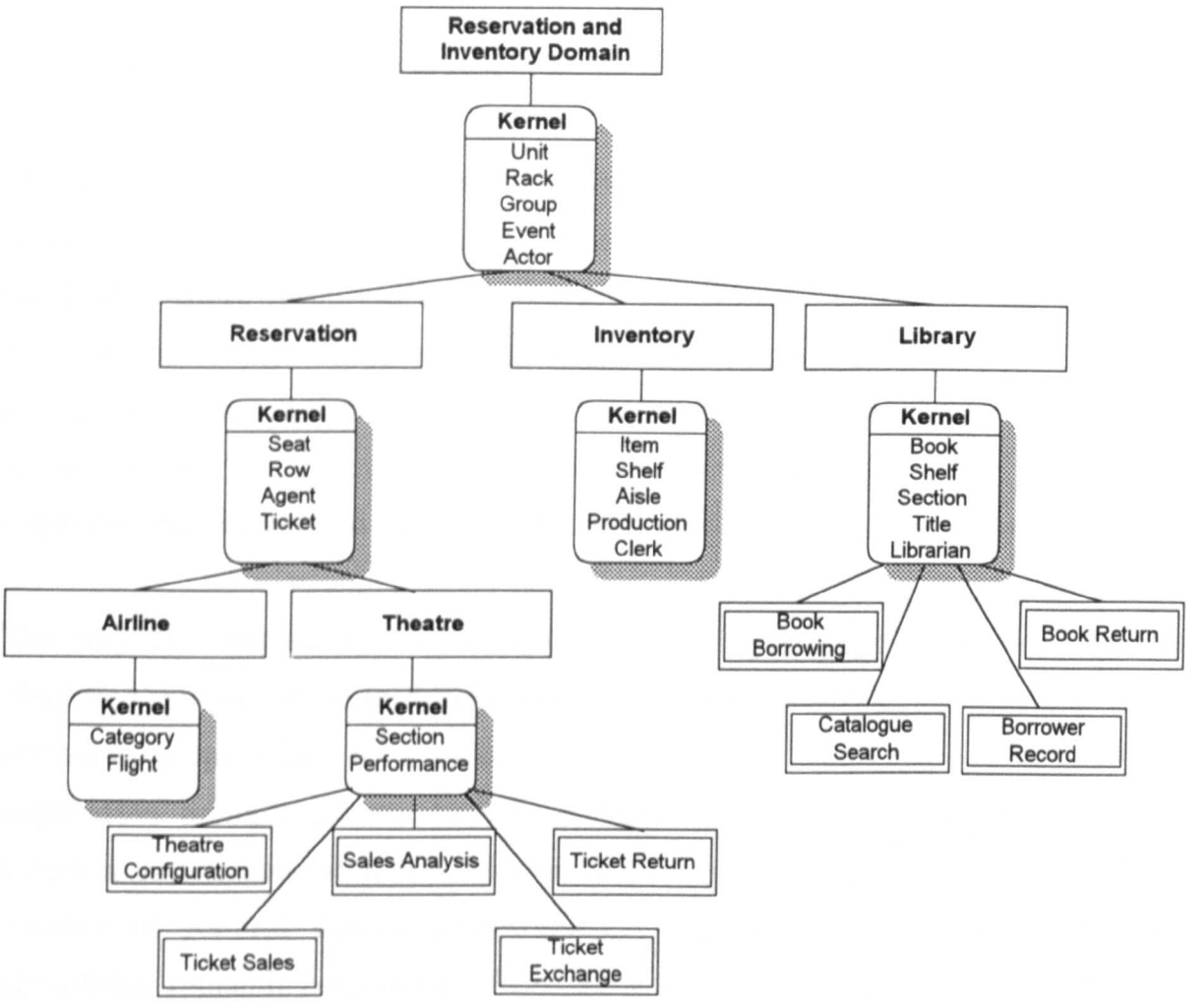

Figure 4-9 The Modified Domain Taxonomy

In Figure 4-9, if a system is built within the scope of the *reservation* sub-domain then the components *Unit, Rack, Group, Event* and *Actor* are intrinsic

components, *Seat, Row, Agent* and *Ticket* are frequently reusable whereas *Category, Flight, Section* and *Performance* are candidate.

4.6 Summary

In chapter two, domain analysis was introduced and it was stated that the process is knowledge acquisition and presentation for the domain resources. In this chapter, we have introduced our approach which is based on domain analysis. The knowledge presentation aspect of the process was emphasised. The idea is to provide the developers of new systems with a structural presentation of the domain. The reusable assets of the domain are organised according to that structure which represents multiple levels of domain scopes.

We have identified three outputs as a result of domain analysis which are called domain resources. These are domain scenarios, rationales and dictionary. We think that these resources are important for building other reusable assets in the domain model. Three more artefacts are generated when the domain is modelled. These are domain taxonomy, reusable components and reference architectures. The resources and artefacts are the elements of the DOOR approach which are used in domain modelling and system development.

The way the domain taxonomy is constructed and reusable components are identified could be done in different ways. It could be conducted in top-down or bottom-up approaches or a mixture of the two. A thorough analysis of the application domain is essential for understanding the domain dependencies before a satisfactory structure can be built. On the other hand we propose a number of general design guidelines for building domain taxonomies and identifying reusable components. We recommend a mix of top-down and bottom-up approaches when taxonomies are built. The exercise should be done in a number of iterations until the final version is reached. This evolutionary feature of the domain model is very important and should continue after the initial version is built. In fact, all the DOOR assets must be evolving as the domain itself evolves.

In the next two chapters the technology (generic architectures) and the process (domain engineering and application engineering) aspects of the approach are described. These also have their effect on the evolution of the domain assets.

Chapter Five

5. Generic Software Architectures and Architectural Models.

5.1 Introduction

In this chapter, the generic software architectures and their architectural models are discussed. The chapter is organised according to the basic elements of the architectures. In section 5.2, we will explain the general model of the generic software architectures and its elements. The architectures are built from a number of architecture schemas using a graphical notation. The structure of the architecture schemas is described in section 5.2.1.

Following is a detailed description of the basic elements of the architectures which are the reusable components, their inter-relationships and constraints imposed on them. Section 5.3 describes the semantics of reusable components within the domain model, and proposes a classification of the reusable components with respect to a number of categories. Section 5.4 introduces the generic architectural models that are used for building reference architectures. Seven generic models, which are based on object-oriented modelling, are described in this section. These models are used to model the relationships between components when systems are synthesised. Two types of models are distinguished according to the components' relationships which are static or dynamic.

In section 5.5, we will introduce an example showing how the generic software architectures are used for building domain reference architectures from reusable

components. In the last section we will conclude the chapter with a summary of the main points covered in the chapter.

5.2 The Generic Software Architectures Model

As explained in the previous chapter, the domain taxonomy presents the domain organisation and the scope of the reusable components on the taxonomy. The generic software architectures provide a technique for modelling the inter-relationships between the reusable artefacts in the domain model. In other words, it models the dependencies between these artefacts in order to form some meaningful design abstractions within the domain. The technique is based on the following principles:

1. The generic software architecture models the domain-specific transactions and design procedures in terms of the reusable components within the domain and the inter-relationships among them. A reference architecture is a representation of its domain knowledge as encapsulated by the reusable components. Therefore, a component may not represent a meaningful abstraction outside its domain scope.
2. The reference architecture contains a number of architecture schemas which are composed from generic architectural models. The architectural models define the reusable components that comprise the schema and the relationships between them. Constraints limiting the components' behaviour are prescribed within the generic model.
3. Using an automated architecture assessor which uses a set of design rules for processing the architecture design. A number of validation procedures are embedded in the assessor for checking the architecture constraints and the validity of the design.
4. A graphical representation of the architecture schemas using a pre-defined notation is used for modelling the components' relationships. Using the supporting tool, the architectures can be built graphically and then passed to the assessor for validating the design.
5. The architectures are organised according to the domain scope which gives access to other assets within the same scope in the domain model.

6. The generic software architectures provide a device for tracing and retrieving reusable components from the architecture schemas. To explain this point, suppose a number of components are linked in one architecture schema, it is sufficient to locate one component in that schema in order to trace the others. This is done by following the links from the first component to the others. The generic software architectures specify what components are required and how they are linked together to construct a useful abstraction.

In order to understand the way the generic software architectures work, we need to describe their basic elements. Like all other architectures, the basic elements are *components, relationships* and *constraints.* In the following sections of this chapter these elements are described in detail by classifying the components into categories and the relationships into a number of generic models. The constraints are modelled by imposing restrictions on inter-connecting different components' categories using different architectural models. In addition, the generic software architectures have a notation for specifying the schemas graphically and an assessor for validating the architecture schemas. Figure 5-1 shows the general model of the generic architectures.

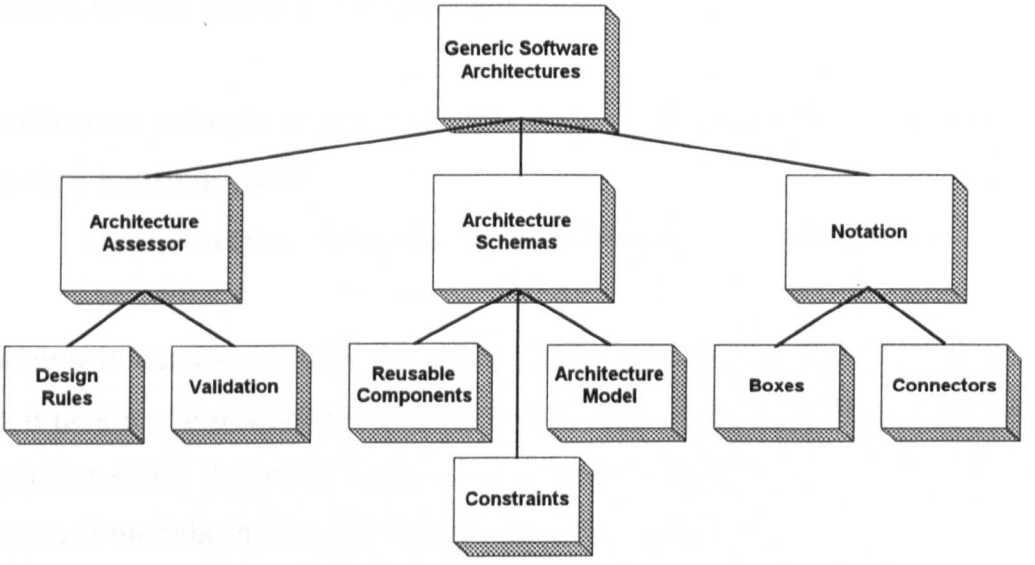

Figure 5-1 The Generic Software Architecture Model

As shown in Figure 5-1, the generic software architecture uses the architecture schemas as the centre of modelling relationships between components. Each schema is specified in terms of an architecture model and two or more components. It also contains a number of constraints imposed by the selected architectural model upon the component types in the schema. Each architectural model is associated with a graphical notation used for specifying relationship between its components. The validity of the architecture schemas is assessed by the architecture assessor. The assessor uses the architectural models constraints for validating the design of the schemas and a number of design rules for assessing the overall reference architecture by locating redundant relationships or suggesting modifications for improving the architecture.

5.2.1 The Architecture Schemas

A reference architecture within an application domain is a collection of design conceptions that can be reused when systems are synthesised. A reference architecture in the reuse context contains one or more architecture schemas that describe the dependency between components which make up the design conception. Thus an architecture schema represents a behaviour abstraction that specifies a reusable design conception in the domain. In some cases architecture schemas define a number of design alternatives to choose from.

An architecture schema is specified by two or more reusable components and a relationship between them.

Architecture Schema = Components + Relationship

The schema is used to specify the dependency between the components in order to form a larger component that represents a domain-specific design abstraction. The relationships between components are specified in terms of generic architectural models that could be used in any domain.

In DOOR, the architecture schemas are the elements of the reference architecture. They are used to model how the components identified in the

domain taxonomy are related. Thus the schemas provide a modelling mechanism for showing how the components could be linked together to form a useful design decision. Some of the schemas provide domain-specific design conceptions and some provide design alternatives from which one could be selected when systems are synthesised.

5.2.2 Notation

An architecture schema is described graphically using a pre-specified notation which is defined in the generic models. Each generic model is associated with a graphical representation that is unique to it. This helps in building the reference architectures and tracing its components automatically.

5.2.3 Architecture Assessor

The architecture assessor is an automatic facility that is used to validate the architecture design. This is done by checking the design against the constraints of its models. In chapter six and seven, we will introduce a set of guidelines and implementations for assessing the design.

5.3 Semantic Description of Components

In the last chapter we described the 3-D model (Behaviour, Scope and Reaction) of reusable components. The model helps in describing the semantics of the components. Basically, each component is described by three elements: its *interface*, the domain *constraints* and the *reaction space*. These are indicated in the three dimensions of the model, the *behaviour, scope* and the *reaction* to its behaviour. The significance of the three dimensions on the semantic description of components is presented graphically in Figure 5-2.

The application domain constraints are applicable to all the other elements of the model. In other words, the domain constraints are imposed upon the component's design, its reaction space and the interaction between the component and the reaction space. Therefore, the model describes the semantics of the component in association with its scope. It does not describe or guarantee certain behaviour abstraction outside the domain scope for which the component is designed.

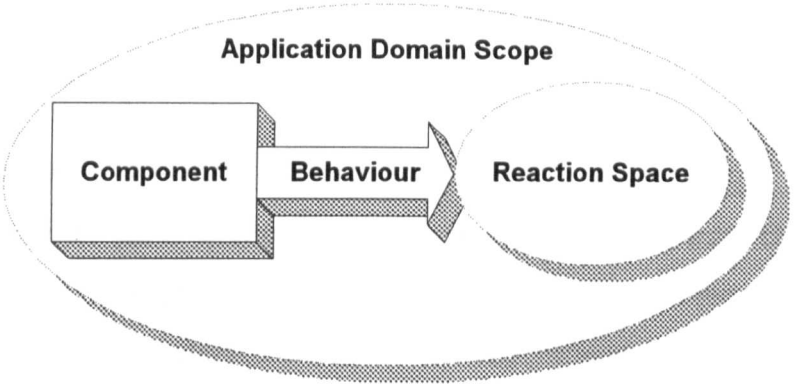

Figure 5-2 Component's Semantic Description

Specifying the component interface is not sufficient to understand its behaviour. We need to specify the interaction of the components with other components in the domain and the constraints imposed on it by the domain.

Some of the constraints are imposed on the way the component is interconnected with other components or by restricting its reaction space (see chapter four for details). For example, specifying a component in a certain scope will restrict it to interacting with the components within its reaction space. Remember such a constraint is imposed by the domain requirements and it is an essential part of the domain knowledge modelling. It is not the choice of the domain engineer to model it this way or the other. As an example, if a certain component is connected to a list of components in an architecture schema such that only one component from the list should be selected, then this schema represents a constraint on the component's behaviour. Again such a constraint is imposed by the domain requirements not by the domain engineer.

Other constraints can be imposed on the component by enforcing a certain implementation or component's type when it is designed. For example a component could be designed as parallel or sequential, passive or active, etc. The component interaction with other components will be restricted by its type when described in the reference architecture. On the other hand the type of the component will tell us more information about its expected behaviour. Once

more, these restrictions are imposed by the domain and are part of the domain knowledge modelling. This is why the 3-D model describes the component's semantics within the domain scope.

5.3.1 Classifying Reusable Components

Components vary in their behaviour, role, scope and the way they respond to messages within a particular domain. This variation affects modelling the relationships and links between them when the reference architecture is built. Different types of components allow different types of relationships. Thus, by describing the component types we can check on the validity of a certain architecture schema, and provide some design guidelines or impose some domain-specific constraints.

Components are classified according to their *scope, implementation* or *mode of operation.* In this section, each of these categories is classified.

5.3.1.1 Classification According to Scope

In chapter four, reusable components were classified according to scope; in this section, we will outline how this category affects the components' retrieveability and traceability.

In this category, components are classified as

- Intrinsic
- Frequently Reusable
- Candidate
- Bounded

Intrinsic components are those which make the core in any system built in the domain. For example if we are modelling the domain of Auto-teller machines, a component like *card_reader* is **intrinsic** because it is essential to any system in that domain to have a card_reader in its design. The reusability of such components are guaranteed (providing, of course, that they are designed as

reusable; see chapter two for details) due to the fact that every system built in the domain requires them as part of its design. Such components are found in the main domain kernel. Once an intrinsic component is identified in the kernel, other components can be located by following the links that relate them to the intrinsic one; retrievability is improved by the availability of intrinsic components in the domain model. If the system is designed within the scope of a sub-domain then components identified in the parent domain kernel are considered as intrinsic components too.

Frequently reusable components are the kernel components of the sub-domain within which the system is built. These components are frequently reusable within their sub-domain scope and are the distinguishing feature of that scope. Therefore the kernel components of the main domain are both intrinsic and frequently reusable within the main domain scope. The kernel components of a sub-domain are important to determine the boundaries of the sub-domain. Specifically, the reaction space of the kernel components denotes the boundary of the sub-domain. Hence, identifying the right frequently reusable components, when the domain taxonomy is built, is essential for building the reference architecture correctly.

Candidate components are less reusable than the frequently reusable ones. These may be reused if linked to some frequently reusable components by an architecture schema. Usually when a system is developed within the scope of a certain domain then components that are found in any of its sub-domain kernels may be reused by this system, hence they are **candidate** for reuse for that system. As an example, consider the case that a certain architecture schema states that a component modelled in a certain domain scope needs to be linked to one of a list of components specified in the domain subordinate. The components in the subordinate are candidates since their reuse is not ensured by that particular schema. The schema is only stating possible reuse of such components.

Bounded components are the least reusable type of components. Their reusability is bounded to specific systems within the domain structure. They are found in the lowest level of the domain taxonomy tree as the unity domains.

5.3.1.2 Classification According to Implementation

Component could be classified according to functionality as

- monomorphic or polymorphic
- autonomous, semi-autonomous or non-autonomous

Monomorphic components provide a single implementation to the encapsulated functionality. In contrast, *polymorphic* components provide multiple implementations for the encapsulated functionality.

The theory behind the polymorphic components for reuse is based on, but goes beyond, polymorphism in object oriented systems. In the O-O paradigm, polymorphism means that the objects may have multiple forms but treated by the compiler as the same type with the ability to locate the right form to execute the right method at run time. This is fine, but in domain oriented reuse we have a situation when a reusable component varies when it is applied in different sub-domains. A polymorphic component is used to provide an implementation support for creating multiple forms of the component. As an example of such a component consider a controller component in process control domain. The controller is shared by a wide range of applications, however according to the application there are different control algorithms which could be used to implement the control function in the component. The best way is to design the controller component as a polymorphic component in that process control domain. Inheritance and software architectures play major roles in the implementation of such components. Later in this chapter, we will use an example to show how polymorphic components can be used to create a comprehensive reusable component within the domain.

Our approach is different from Booch's forms of reusable components, which are multiple variations of the same component [Booch 1987]. The variations in our approach are achieved by the actual application of the component in the proper sub-domain which means that the component forms are bound to their scope of reuse. It also reduces the amount of redundant code that is duplicated in the implementation of Booch's forms. Finally, polymorphic components share the same interface.

Going back to the monomorphic components, some components are designed as monomorphic for one good reason which is eliminating confusion about the usage and the scope of reuse of a certain component. As an example, components that are safety-critical and where multiple implementation can cause non-deterministic behaviour or ambiguous response; such components are best designed as monomorphic. In such cases, the domain requirements (safety-critical) restricts the implementation of the component to monomorphic implementation even if it was possible to design it as polymorphic.

Reusable components can also be classified according to their autonomy levels as **autonomous, non-autonomous** or **semi-autonomous.** By autonomy we mean how dependent the component is on the existence of other components in the reference architecture. An autonomous component is defined as a component that provides complete abstracted services without depending on the existence of a particular component, whereas a non-autonomous component is a component that is contingent on the existence of another autonomous component for it to be accessed. Usually a non-autonomous component is linked to an autonomous component as 'part-of' the autonomous component (see section 5.4). As an example, if we go back to the *controller* example, a *control-algorithm* component may be used if and only if the *controller* component exists in the system. So the *controller* component is autonomous while the *control-algorithm* is non-autonomous and the *control-algorithm* component is modelled as 'part of' the *controller* component. Another feature of the non-autonomous components compared to the autonomous components is that a non-autonomous component

uses the interface of its composite component. In this case, its behaviour will be a subset of its composite component's behaviour.

Another case occurs when the composite component's behaviour is not complete without the availability of the non-autonomous component. In this case, although the composite has its own interface which makes it possible for other components to access its services, its behaviour is contingent on the non-autonomous component. The composite component, in this case, is semi-autonomous. The combination of the two components, however, is a broader autonomous component.

5.3.1.3 Classification According to Mode of Operation

When components are put into operation, they take several forms as

- active or passive
- concurrent or sequential

An **active** component is basically a component that does not need to synchronise its execution with other component's execution. Once they are created they run to completion. They do not depend on the results or the execution of other components. Their life-span could last as long as the system's life span or only for a shorter time. Examples of such components can be found in every domain. In the process control domain, for instance, a timer component is active component where it keeps running as long as the system is in operation, generating time signals to synchronise the operations of other parts and components of the system.

On the other hand, a component could be described as **passive** which is normally at idle state. Such a component only comes to life when triggered by another component to provide a service. When the service is accomplished, the component returns back to its original state waiting for another trigger to activate it (see section 5.4.5). A typical example of such components is a print manager component. The component provides its services to any other

component which requires a print service regardless of their type, state or any other considerations.

A component could also operate as **concurrent** or **sequential**. Sequential components preserve one thread of control. This means that the component is created to serve a certain client. If another client is requiring a service of the same component another instance of the component should be created for that client. Concurrent components, on the other hand, allow multiple threads of control to be active at the same time within the component behaviour. It is the responsibility of the component itself to solve the problems that are usually associated with parallel processes and the client should not take any responsibility to deal with mutual exclusion problems or other problems.

This view to the component concurrency is different from Booch's view [Booch 1987]. In his view, components are classified as **sequential, guarded, concurrent** or **multiple**. The last two forms are parallel processes where the mutual exclusion is enforced by the component itself. The difference is that multiple components allow multiple simultaneous readers while concurrent components sequentialise client access to the component whether they were readers or writers. We feel that this classification is over-presenting the concurrency situation of the components, especially we are dealing with reusable components where the internal implementation of components are restricted by the domain constraints and the scope of applying the components in the reuse process. On the other hand, the first two forms of Booch's classification are actually both sequential. Guarded components break the rule of providing a comprehensive interface to their application and require the client to take the responsibility of providing a front-end device to enforce mutual-exclusion. This is a serious breach of the rules for uniform handling of resources inside a reusable component which hinders the quality and reliability of reusable components.

In our approach, the implementation of solutions to a particular problem in the domain is bound to that domain constraints, and therefore the solution is part of

the component semantics. This means that the component exhibits a complete and cohesive abstraction. Reusable components should not require any support of an outside entity to accomplish a primitive operation and should allow the user to decide how a particular problem should be solved. If multiple solutions to a problem are required then the component is better designed as polymorphic.

5.4 Generic Architectural Models

In Generic Software Architectures, the domain knowledge and component dependencies are modelled using a number of generic architectural models. These models are descriptions of standard relationships that interrelate components in the domain model. The choice of these models is made to represent common object-oriented relationships between components. Some of these relationships have direct object-oriented implementations such as inheritance. Others are not supported by implementations; however, they are commonly used in object oriented systems such as the event-driven model.

The models are described in terms of the components' types that are related and the relationship between them. One or more architectural models are used to build one architecture schema which specifies a meaningful abstraction in the domain scope. Choosing certain models for describing an architecture schema means specifying particular relationships and imposing some constraints on the components' types that are used in the schema. Also, a certain component with a particular type is restricted by its type with regards to the architectural models (relationships) that could be chosen for describing the design conception.

The use of the generic architectural models offers the following benefits:

1. Because they are generic these models can be used for modelling different domains. It also means that the experience gained in modelling one domain could be reused in modelling other domains.
2. Presentation-wise the domain is modelled in terms of technical relationships that are familiar to software engineers. In DSSA's, the domain architectures

are built using domain-specific parts and terminology that are alien to software engineering practitioners.

3. The reference architectures built using the generic software architectures represent domain knowledge in terms of reusable software components. This offers the software developer a quick transition from the problem space into the solution space allowing for the reuse of design knowledge within the domain.
4. The generic architectural models provide alternative ways for linking software components together in order to form bigger reusable components that represent broader domain abstractions.
5. Using architectural models standardises and automates the process of building reference architectures. This makes the building and validation of the reference architectures more systematic.
6. There is a possibility of transforming the architectural schemas into detailed design patterns of interconnected objects which makes reaching a solution for the developed system even more systematic.

In this section, we will discuss the generic architectural models that are used to describe possible ways for linking components together in a reference architecture. Each model comes with four parts; the type of the components in the model, a relationship that links the components, a graphical notation to present the model and a description of the constraints imposed by choosing that particular model.

The generic architectural models are divided into two categories; static relationships and dynamic relationships as listed below:-

- **Static Relationships**
  - Generalisation-specialisation model
  - Whole-part model
  - Association model
- **Dynamic Relationships**
  - Client-server model

- Implicit creation model
- Event-driven model
- Batch-process model

Before discussing the different models, it is useful to explain the roles of the constraints in this technology. Constraints are used for restricting the use of the architectural models to specific component types for the following reasons:

1. When components are identified their types are specified to comply with the domain rationales. When the components are used in the architecture schemas there are certain types of components that could be used for a certain model. The constraints specify the cases of component types which could be used with a certain model.
2. In some cases, when components are linked using the architectural models the constraints specify design alternatives relating to how these components could be applied when systems are synthesised.
3. The constraints are used for validating the architecture schemas in the domain model. They also are used for modelling the domain-specific constraints by choosing specific component types in the schemas.

5.4.1 The Generalisation-Specialisation Model

In this model components are linked by inheritance. The inherited component is called the *parent* component and the inheriting component is called the child component. The model links the components by the **is-a** relationship which is described by the schema in Figure 5-3-A

Although inheritance is used for different reasons in the object-oriented paradigm, here the is-a relationship is used to model the *generalisation-specialisation* case. This is because this is the most common case for reuse by inheritance. Hence, in the above schema **Child** is a special case of **Parent**. The child component inherits the services, attributes and type of the parent component. If, for instance, the parent component is not declared as concurrent then its children cannot be designed as concurrent, because the interface of the

parent component is not designed to allow such behaviour. There is another reason with regards to reusability; increasing understandability of the components in the domain infrastructure. If a component is a special case of another component then it should bear the same attributes as the general case component except for its specialisation feature.

The child component may provide its own services in addition to the parents services. In the reuse context, the child component may be modelled in the kernel of a sub-domain of the parent component domain. The parent component constitutes a meaningful abstraction where it can be reused without requiring the existence of the child component. The child component specifications include only the specifications of the added features.

If the parent component is polymorphic, the child component can re-define some or all of the parent components. Linking the child component to other components within its sub-domain assumes that the polymorphic services are only provided through the child which over-rides those polymorphic services. If any service is required from the parent component then a separate link should be drawn to the parent component. A separate object is instantiated for both cases even if the implementation language allows such reference through the child component (as in C++).

A special case occurs when the parent component provides an interface to its services but does not provide implementation to one or more of those services. The implementations of the deferred services are provided by the children components. Such a component is called *Abstract Component* and must be declared as semi-autonomous type.

If the child component inherits from more than one component then it will be described as shown in Figure 5-3-B. In this case the child component shares the services of both parents. The parents' and child's type is restricted as described in the following constraints.

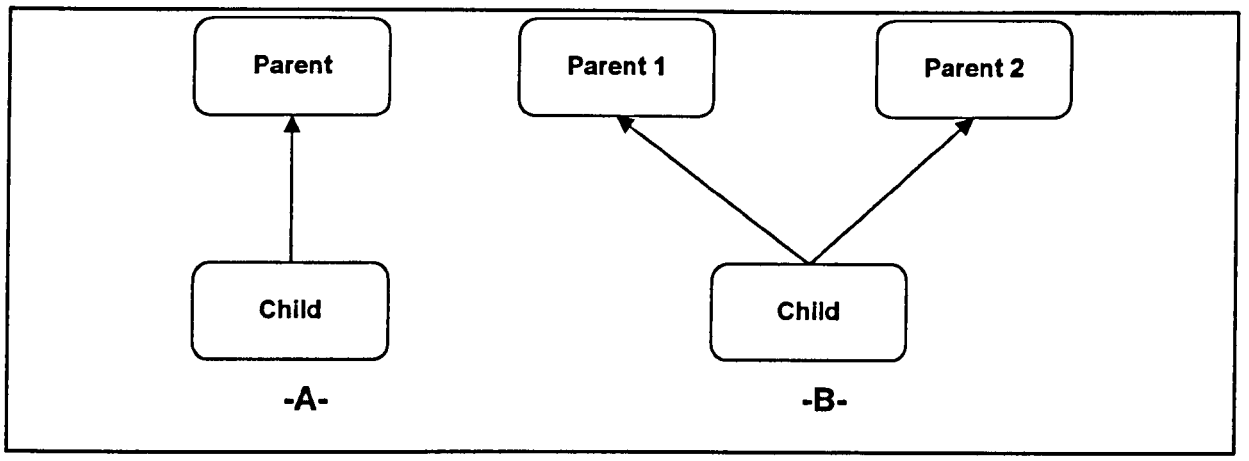

Figure 5-3 Generalisation-Specialisation Generic Model

Constraints (Generalisation-Specialisation)

1. The parent and child components must be of the same type (active, passive, etc.). The only exception is when the parent component is polymorphic then the child component may be monomorphic.
2. If a child component overrides one or more of the parents services then the parent component must be polymorphic.
3. If the parent component is an abstract component then it must be declared as a semi-autonomous, polymorphic component. The semi-autonomous type means the behaviour of the parent component is contingent on the behaviour of its children.
4. If the parent component is polymorphic, unless it is an abstract component, a link to one of its polymorphic services by an architecture schema is explicit to its own implementation. If the message is bound to the child's implementation then a separate link to the child component must be established.
5. If the domain knowledge requires that the behaviour is not determined and could be decided upon in later stages of the design then the parent component must be designed as an abstract component.
6. If the child inherits from two parents then the parents (as well as the child) must be of the same mode of operation (active or passive and sequential or concurrent). Both parents and the child may either be polymorphic or

monomorphic. If the parents are of different levels of autonomy (autonomous, semi-autonomous or non autonomous), then the child inherits the lowest level.

5.4.2 The Whole-Part Model

In this model, components are linked by the **aggregation** (is-part-of) relationship. Unlike the generalisation-specialisation model, the whole-part model integrates two or more components to build one autonomous component. One of these components holds the rest and provides a mechanism to manage their access and operations. This component is called a **composite** component and the managed components are called **agents.** The agents are non-autonomous components that rely on the presence of the composite to invoke their services.

The composite component is autonomous and may be accessed by other components in isolation of its agents. However, if a certain service within the composite component requires one or more agents to accomplish its functionality then that service will be contingent on the existence of the relevant agents. In this case the composite component is semi-autonomous which means that it can only provide that particular service when the relevant agents are available. However, the broader component resulting from the aggregation relationship is autonomous.

The aggregation relationship is described by the schema in Figure 5-4-A. This schema states that *'Composite'* contains both *'Agent1'* **and** *'Agent2'*. The **and** means that the presence of both agents Agent1 and Agent2 are essential for the behaviour of 'Composite'.

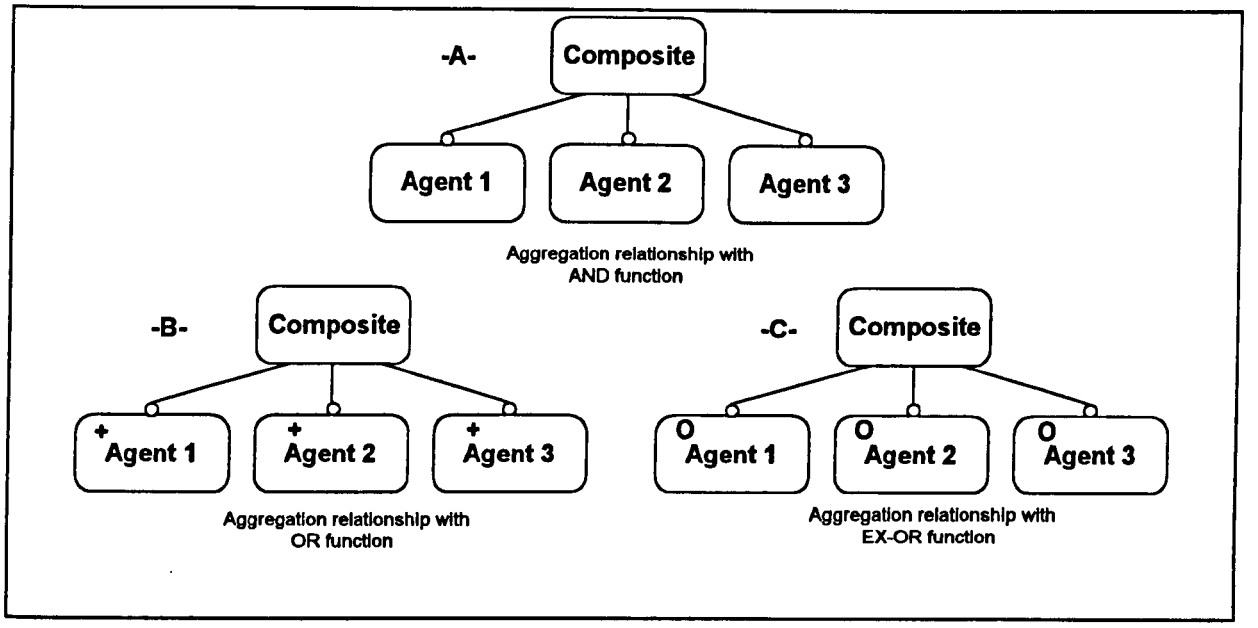

Figure 5-4 The Whole-Part Generic Model

If the semantics of the resulting component requires any one or more components from a list of agents then the aggregation is described by an **OR** function as stated in the schema in Figure 5-4-B. This is a true **OR** function where any one or more of the agents could be aggregated by the composite.

Another case exists where the resulting component after aggregation provide a selection of different cases. Each case is supplied by one agent and the container can only hold one selection at a time. In such a case an architectural description should allow an **EX-OR** function as shown in the schema in Figure 5-4-C. In this case *'Composite'* contains a choice of any agent (but no more than one) appearing in the list.

There is no restriction against constructing a multi-layered aggregation relationship between components with the possibility of having different style for different layers. Figure 5-5 shows an example of a layered aggregation schema.

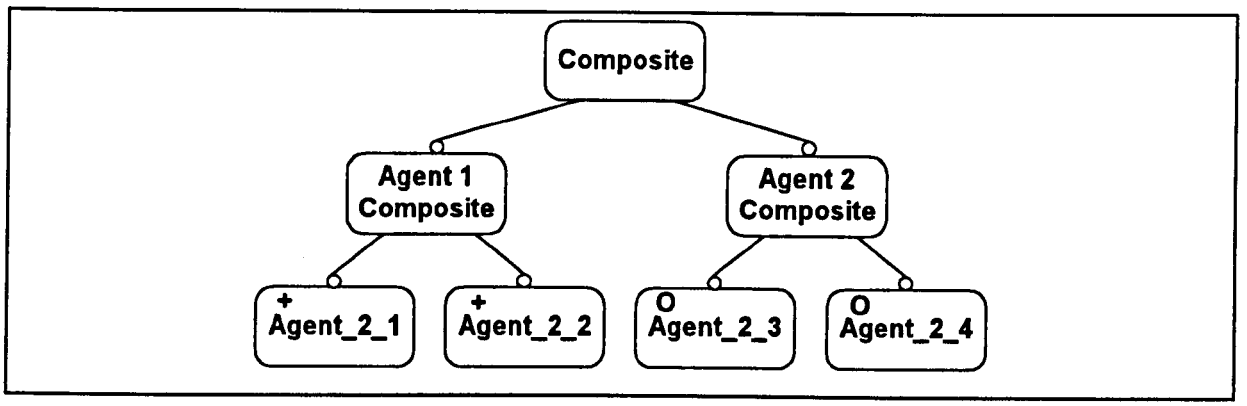

Figure 5-5 Multi-Layer Aggregation Relationship

Constraints (Whole-Part)

1. The composite component must not be a non-autonomous component.
2. The agents could be semi-autonomous or non-autonomous. However, if the agent itself has its own agents, then it must be semi-autonomous.
3. When the composite component is bound to the aggregation of one or more agents then it must be declared as semi-autonomous. Specifically, if the aggregation relationship has an AND function then the composite must be declared as semi-autonomous.
4. If the aggregation relationship has an OR or EX-OR function and the composite component is declared autonomous then an extra fictitious agent is assumed in the description. The fictitious agent component is called an *Empty* agent which implies that the composite component does not depend on the existence of any of its real agents. If the composite component is declared as semi-autonomous then at least one of the agents in the list must be aggregated in the composite component.

5.4.3 The Association Model

The whole-part model assumes that the agent component behaviour is valid only within the behaviour of the composite component. Sometimes composition is used to achieve reuse without any contingency between the composite and the agent component. In other words, both components are autonomous and each one is an agent of the other. This is modelled as association between the two components.

Association is a principle used to manage complexity and is used to imply that two components happen at the same time or under similar circumstances [Coad an Yourdon 1991]. The model does not place any contingency on the relationship between the two components in terms of aggregation or inheritance. Each component needs to be aware of the other's abstraction in order to accomplish its own behaviour. The association relationship (sometimes referred to as instance connection) is described in the schema shown in Figure 5-6:

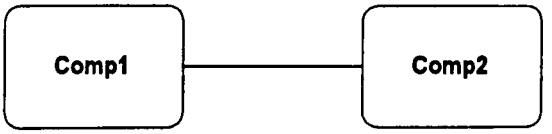

Figure 5-6 The Association Generic Model

<u>Constraints</u>

1. The two components must be of the same autonomy level. For example they have to be both autonomous. If they are both non-autonomous then both must be described in the same aggregation relationship as shown Figure 5-7

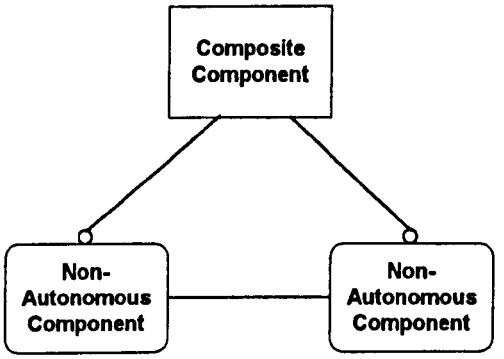

Figure 5-7 Non-autonomous Components in The Association Model

2. If one of the components is autonomous and the other one is semi-autonomous then they can be associated to each other if the semi-autonomous component has at least one agent in AND-function aggregation relationship (see Figure 5-8). This condition guarantees that the combination of the semi-

autonomous component and its agent is an autonomous component and then this will satisfy the first constraint.

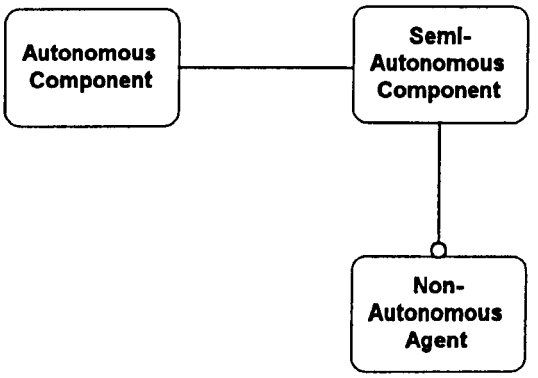

Figure 5-8 Semi-autonomous Components in The Association Model

5.4.4 The Client-Server Model

This model is described by two components; a **client** and a **server** which are linked by a **message** relationship. The messages is passed from the client to the server. The server takes control and executes a method to service the client's request. The results of the invoked method are returned back to the client at completion of the service. A message relationship is described as shown in the schema in Figure 5-9. In this schema where **Comp1** is a *client* of **Comp2,** the *server* and **service_id** is a reference for the requested service; this could be in the form of the method name or an ID number.

The nature of the service is captured in the interface of the server component and the functionality of the requested method. The client component transfers control to the server and enters a wait state until the completion of the invoked service. The server then returns control to the client component and the client resumes its operations.

Constraints (client-server)

This model imposes the following constraints on the design:-

1. The server component must be *Active* and *Autonomous* type component.

2. The client component must be in an *Active State*. Note that an active type component is always in active state; a passive component, however, is normally in a passive state unless triggered by an external condition.
3. There should be no restriction on the number of clients that can access the server.

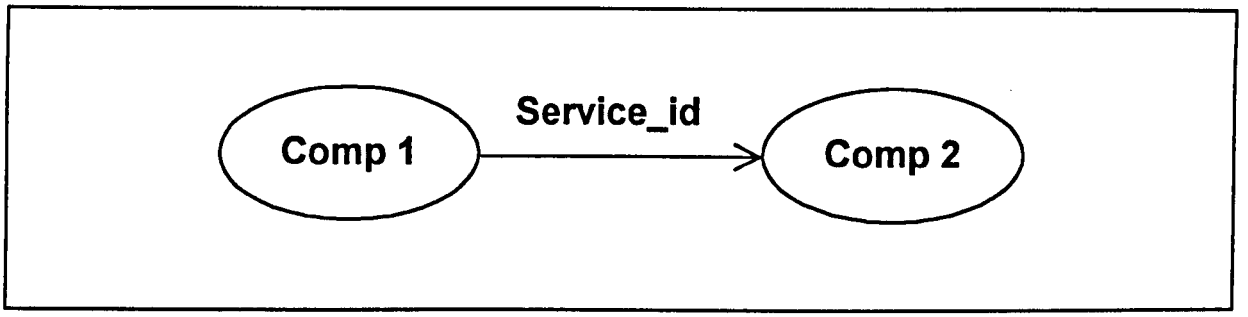

Figure 5-9 The Client-Server Generic Model

If a service is provided by a server for multiple clients then the message can be described as shown in Figure 5-10.

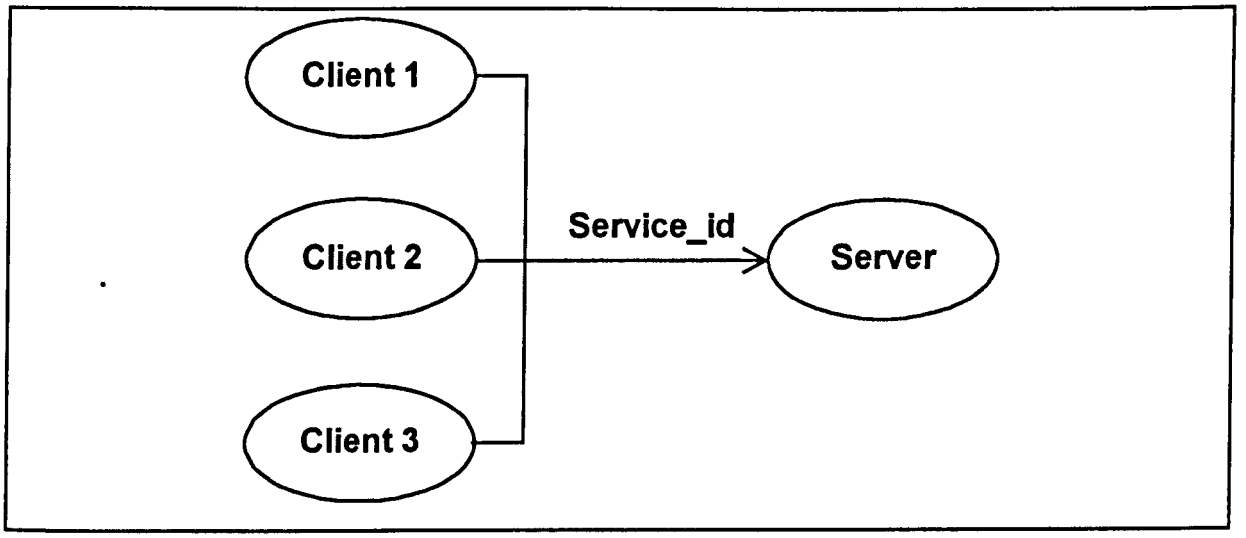

Figure 5-10 Multiple Client-Server

The semantics of this schema is described in the following constraint:

Constraints (multiple-client)

1. All the client components and the server component appearing in the schema must be included in the design when the service is needed. This case is used for modelling a number of components that access the same server with a relation between them such that the change in state of the second component depend on the change in state of the first and the third depends on the second, as explained in the next point.
2. The service must be accessed in the same order of the schema such that the first client gets serviced before the second and the second before the third and so on. The last client in the list must be serviced first before the first one can request the same service again and the whole cycle is repeated.

If the server component is a concurrent one and the clients are expected to call the required service without synchronisation such that the service is provided as a parallel task then the message is described as a concurrent message. A client component can be of any type already described in the previous section. The server, on the other hand, is not allowed to be a passive or a non-autonomous component. Figure 5-11 shows a typical concurrent message schema.

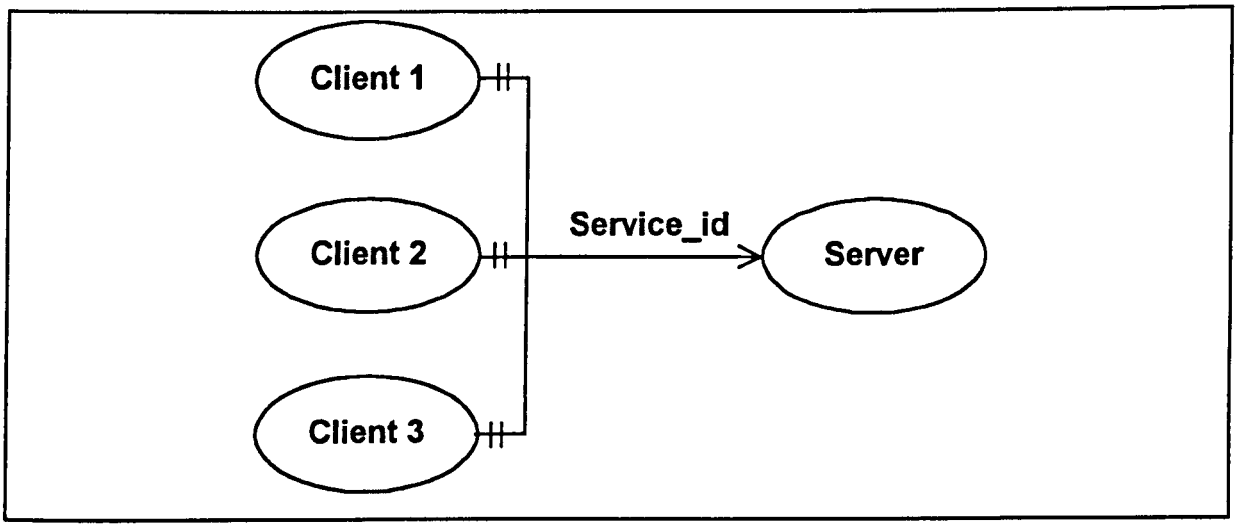

Figure 5-11 Concurrent Client-Server

Constraints (parallel-server)

1. The server must be a *Concurrent* type component. The server must not be a *Passive* or *Non-autonomous* component.
2. All the client components and the server component appearing in the schema must be included in the design when the service is needed. However there is no restriction on the sequence or priority of access among the clients' requests.

5.4.5 The Implicit Creation Model

In this model, components are linked by a relationship called a **trigger**. This relationship requires that at least one of the components is active which is the component that initiates the *trigger*. The other components are passive. The triggered component does not wait for the trigger to arrive but instead it is initially in a (passive) state and when the trigger is initiated this component transforms to an active state. The trigger-initiating component does not require a reply from the triggered component as a result of the trigger processing. However it should have an idea of the time and reaction of the behaviour of the other component. When the triggered component completes its job it goes back to its original state.

In terms of object-oriented implementation the passive component could be implemented as a class that is not created (instantiated) normally. It is created when it is required by the trigger initiating component. The component is then said to be active and continues to be active until it finishes the services that are required by the trigger. When the services are accomplished the component goes back to its original passive state. In terms of implementation, this is done by the component destroying itself and when the trigger is next issued the component is created again.

The passive component is 'implicitly' created at the same time as it is passed a message invoking a particular service by the active component. There is another alternative for implementing this model which is by dedicating a separate active

component for the activity of creating and destroying passive components when they are needed. The schema in Figure 5-12-A shows how a trigger relationship is modelled. In this schema *'Comp1'* is the trigger-initiating component and *'Comp2'* is a passive component. *'Trigger-id'* represents the nature of the message initiated by the active component.

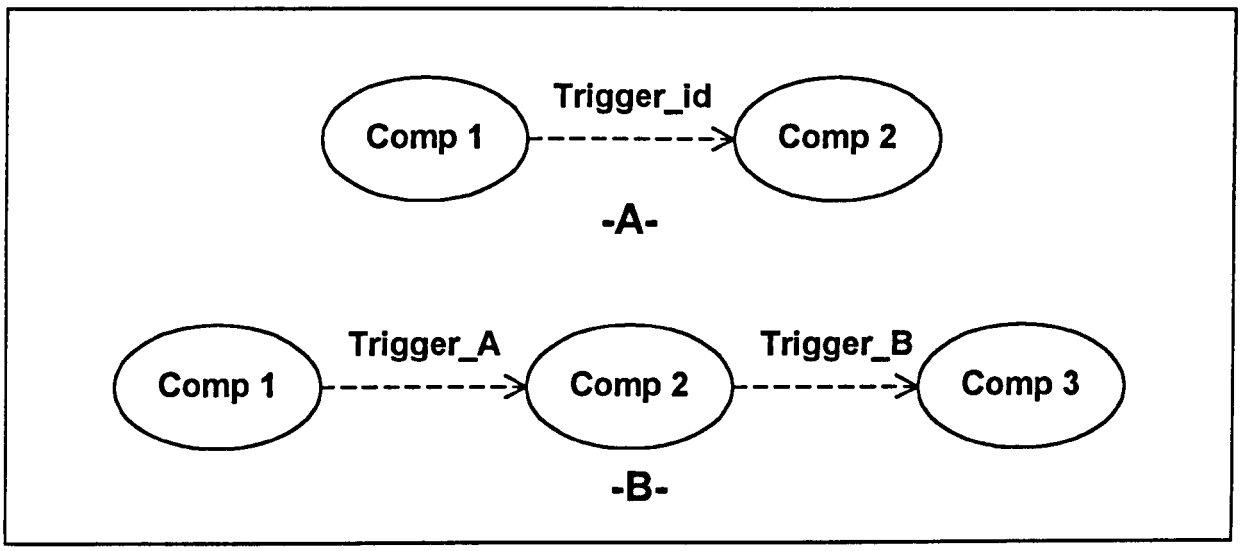

Figure 5-12 The Implicit Creation Generic Model

The trigger should be initiated by an active component or an external source. However a passive component could be allowed to initiate triggers to other passive components if it was already transformed into an active state. The restriction imposed on this situation is that a passive component must only initiate triggers within the same description where it is triggered by an active component. Such a case is described in a schema as shown in Figure 5-12-B.

Constraints

1. The first component that starts the train of triggers must be an active component. A passive component can trigger other components only after it has already been triggered and transformed into an active state.
2. When the triggered component finishes executing its service it must destroy itself and goes back to passive state. This constraint is very important for the

functionality of the component that issues the trigger. Thus it is the passive component responsibility to switch itself to its original state, freeing the other component from managing the state of the passive component.

3. If there are a number of passive components linked in a train of triggers then any component must stay active during the time all the subsequent components in the description are executing their own services. In other words the sequence of creation and destruction of components should be first created last destroyed.

5.4.6 The Event-Driven Model

In this model, services, in a certain component, are not invoked directly by other components. Instead, a component can announce (or broadcast) one or more events. Other components register an interest in a certain event by associating a service with the event. When the event occurs the system itself invokes all of the services that have been registered for that event.

Because this model is aimed at modelling relationships between OO components, we have to make the following assumption. The components that announce the events must explicitly include in their interfaces a method for announcing each event they are responsible for issuing. This is crucial assumption to the behaviour of the components as well as to modelling the architecture connections.

The services that are registered for a certain event are not restricted to be invoked by that event only. In other words, these services are available to other components to access by a client-server model. The event-driven model is described as shown in the schema in Figure 5-13:

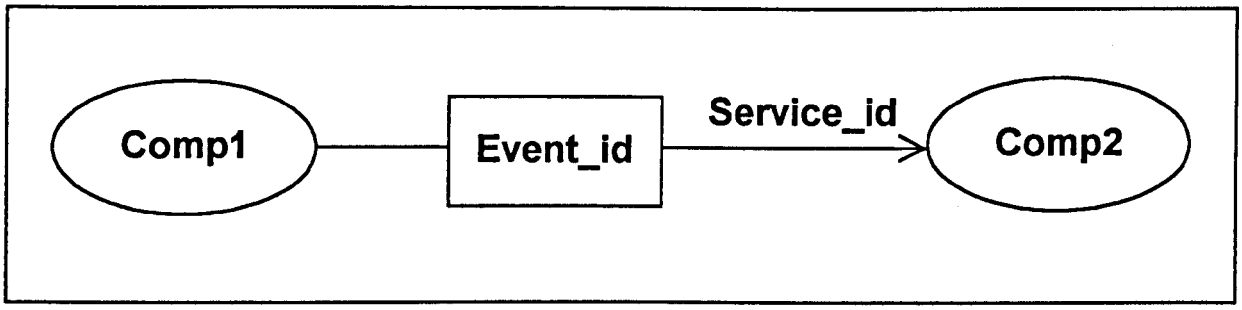

Figure 5-13 The Event-Driven Generic Model

The previous schema states *'Comp2'* is registering *Service_id* to *Event_id*. There is no direct linkage between *'Comp2'* and *'Comp1'*. *'Comp1'* does not know which components are registered to its events and it is not responsible for invoking the registered services when the event occurs. On the other hand *'Comp2'* needs to know about the event that 'Comp1' is broadcasting.

Constraints

1. The component that registers its services for an event must be active and autonomous.
2. A semi-autonomous component may register for a certain event if the semi-autonomous component has at least one agent in AND-function aggregation relationship. This condition guarantees that the combination of the semi-autonomous component and its agent(s) is an autonomous component and will satisfy the first constraint.

5.4.7 The Batch-Process Model

The best examples of a batch process model are the UNIX pipes and filters. The process is passed through a number of components. The components are linked by a front-end coupling; that is the output of the first component is passed as an input to the second and so forth. This requires that the output of the preceding component and the input of the succeeding one are compatible.

The batch process model is described by the relationship **pipeline** as shown in the schema in Figure 5-14:

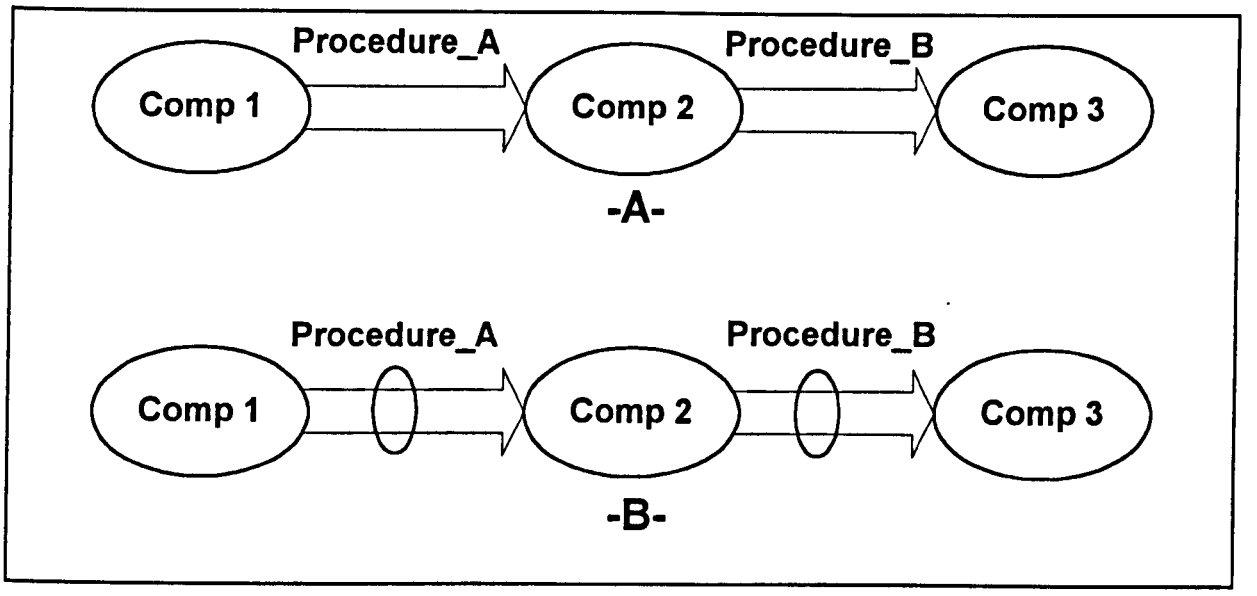

Figure 5-14 The Batch Process Generic Model

The batch process can be modelled in two forms. The first is when the batch job is processed once only and the second when it is a periodic process. In the first model the components receive their inputs from the preceding ones and carry out the operations without having to synchronise tasks with other modules, the result is then dispensed onto the pipe (Figure 5-14-A). In the second model, the pipeline runs periodically passing information between components continuously until the source stops pumping information onto the pipeline (Figure 5-14-B). Normally, the end of a periodic batch process is indicated by an event that could be issued by one of the batch process components or by an external component.

Unlike other relationships between reusable components, the pipeline requires the following considerations to be taken into account:

- How is a pipeline established between components?
- Should a component design be changed to allow a pipeline access?
- When do the pipeline operations start and end? Do we need identifiers for this purpose?
- Where do concurrent components fit in the pipeline?
- What is the best way to pass the data between the pipeline components?

These questions are answered depending on the way pipelines are implemented. Although it is out of scope of this document to specify the implementation of a certain relationship, we feel that it is necessary to provide a design view to answer some of the above questions.

Our idea for synthesising systems based on the batch process model is based, in general, on providing a common information storage between components. The idea is to integrate the components without having to give any attention to their underlying semantics. The problem (as we see it) is a pure interfacing task. The sending component writes to the common storage area and the receiving component reads the information from that area when it is ready to process it.

The pipeline could be established by creating a virtual machine that controls the data traffic between components. The virtual machine is responsible for creating the common data storage; this could be in a form of a file or a memory space on the system's main storage. Once this area is created then the pipeline is actually established and it is now the responsibility of the virtual machine to organise the logistics of the process. This includes determining the start and end of the pipeline process, the synchronisation between components and the handling of memory management and garbage collection.

The other issues like the suitability of the components for pipelines, concurrency and writing and reading to and from the pipeline, are left to be decided locally inside the component itself. Nevertheless, there are two ways to solve this problem. The first is to design special components for the batch process model (e.g. filters). In this case other components cannot be linked using this relationship unless they are re-designed for this purpose. The second approach is by providing abstract classes within a sub-domain as a driver for pipeline components. The abstract-class is a general class whose sole objective is to provide all the operations required in a pipeline component. Any component required to be added to the pipeline should be linked to it using the (is-a)

relationship and this way the component can handle the pipeline through the abstract-class. The abstract class should not affect the behaviour and semantics of the original component. For example if the component is concurrent then the abstract class should preserve its concurrency. That means, while the component is responding to the pipeline it must carry on servicing other clients when required in a normal parallel form.

To summarise the previous considerations as well as taking into consideration other situations, the following constraints are imposed upon this model.

Constraints

1. All the components in this model must be active and not non-autonomous.
2. Every time a batch process starts, the sequence cannot be reverted and must run to completion.
3. If there is a shared storage between the components for passing through data, then this storage area cannot be accessed by other components until the batch process is finished. The data is not valid until the batch process is terminated
4. Normally, the beginning of a batch process is triggered by a message passed from another component, which should be in an active state, to the first component in the model.
5. When a batch process is in periodic mode then the end of the process may be determined by an event issued by another component. This may be implemented by having the last component in the batch process registered for that event. When the event is issued, a batch-terminating service (within the last component) is invoked which causes the batch process to terminate.

5.5 Example of Architecture Modelling

In this section, we show examples of components inter-relationships using generic architectural models. The examples used here are drawn from the reservation systems domain which was introduced in chapter 4. In these examples we illustrate how reusable components and generic architectural

models are used to model design conceptions within the domain scope and impose constraints on the use of these conceptions. In this example, we only show examples of components dependencies within the domain reference architecture. The overall architecture is found in the next chapter. The following components are used in this example, *Rack, Row, Shelf, Unit, Item, Seat, Book, Theatre, Section, Agent,* and *Sales_analysis.*

We show two types of relationships between these components; static and dynamic. In static relationships, we model how the components are arranged in terms of inheritance and aggregation. In dynamic relationships, we show an example using implicit creation and client-server models.

Static Relationships

As described in chapter 4, the similarity between the components are modelled through inheritance using abstract components that are shared between all the sub-domains and inherited by the sub-domain components. The first step is to determine the components' types and then relate them together using the generic architectural models. Table 5-1 shows the components' types.

Table 5-1 Components type table

Component	Type	Component	Type
Rack	Polymorphic, Semi-autonomous	Book	Monomorphic
Row	Monomorphic, Semi-autonomous	Shelf	Monomorphic
Unit	Polymorphic, Semi-autonomous	Agent	Active
Item	Monomorphic	Theatre	Active
Seat	Monomorphic	Sales_analysis	Passive

According to the constraints imposed by the *Generalisation-Specialisation* generic model it is easy to distinguish the abstract components in the table. Abstract components should be declared as polymorphic and semi-autonomous (Constraint 3). Therefore, the two abstract components in Table 5-1 (whose features are used by the four sub-domains) are inherited by concrete components within the domain subordinate. Figure 5-15 shows the use of *is-a* relationships to link the components in the domain. The figure clearly illustrate relationships between components in parent domain kernel (Rack and Unit) and the other components in the sub-domains kernel (see figure 4-9). Since *Unit* and *Rack* components are abstract components, their features are reused by the sub-domain components and may be over-ridden.

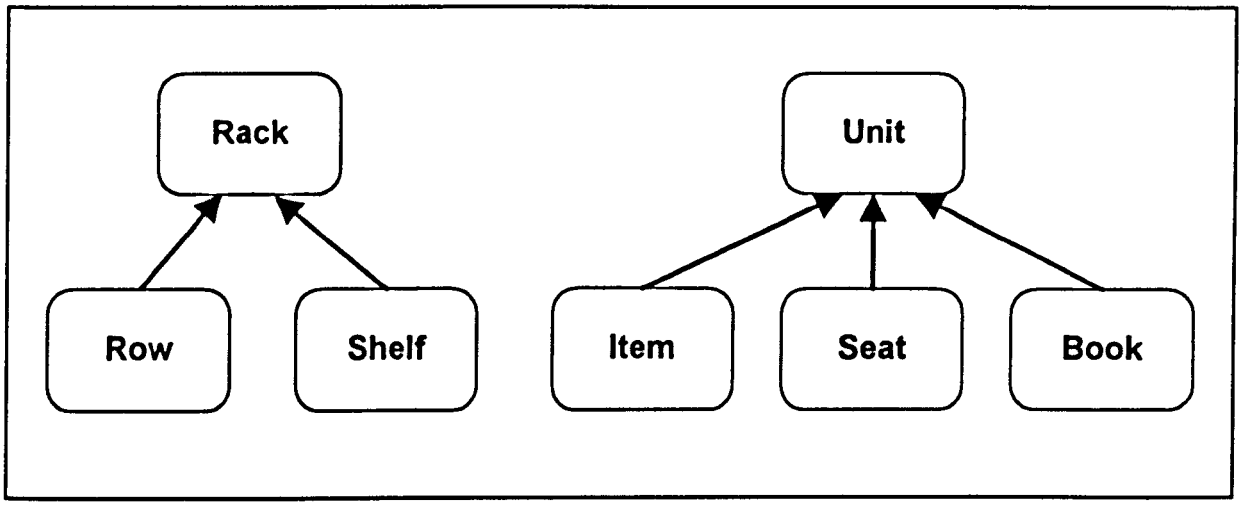

Figure 5-15 Example of is-a Relationship

In Tracz's paper [Tracz 1995], the theatre example is introduced with a structure where every row in the theatre contains a number of seats. This structure is also true for all four sub-domains in the taxonomy shown in figure 4-9, and therefore the same architecture could be reused. The generic architectures have the ability to model this structure in several ways. There is, of course, the way used by Tracz in his paper where each situation is modelled separately as the case of the theatre rows and seats (see Appendix A). One way is to use the abstract components only in the structure using the *Whole-Part* generic model as shown in Figure 5-16-A. In this case the schema models a design conception which can be reused over the four sub-domains.

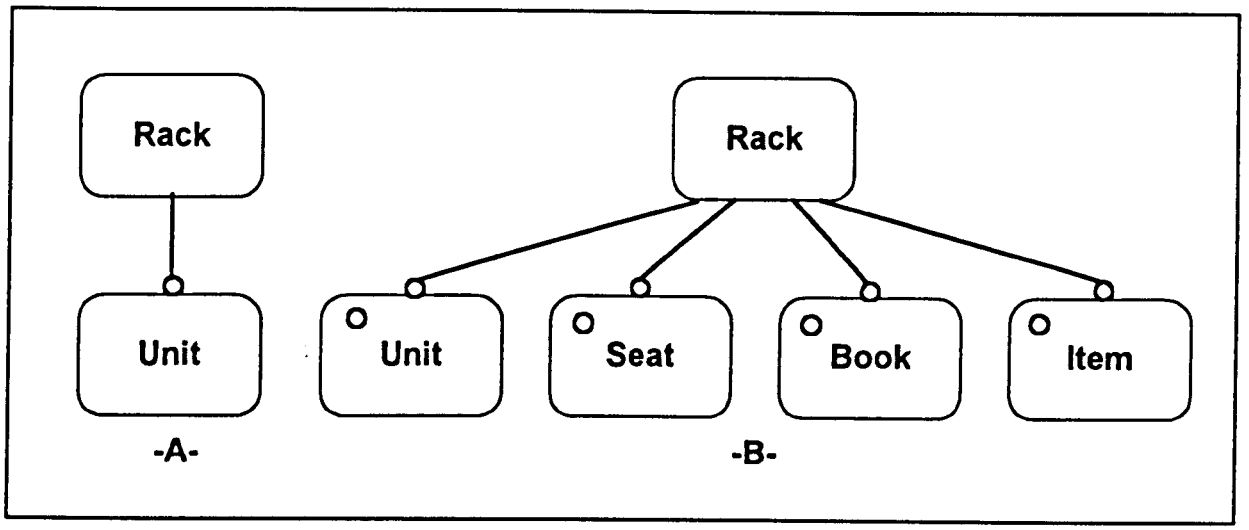

Figure 5-16 Example of Aggregation Relationship

Since both components are abstract components then the schema could be reused in any sub-domain by replacing them by the relevant concrete components. The other way to model it is to use a combination of abstract and concrete components in the *Whole-Part* generic model as shown in Figure 5-16-B. In this case, the schema uses the polymorphic component *(Rack)* and an EX-OR aggregation for modelling a number of design alternatives in the domain including the one shown in Figure 5-16-A. According to the constraints of the *Whole-Part* generic model (constraint 4), since the composite component *(Rack)* is semi-autonomous only one of the components in the schema must be available in its structure. This is exactly what we try to model in the reference architecture. The choice which design alternative to choose, when systems are synthesised, depends on the scope of the system. If the system is developed within the theatre or airline domains then we will choose the *Seat* component, and we will choose *Book* if the scope is the library domain.

Dynamic Relationships

We use the *Sales Analysis* scenario to identify dependencies between some components. The sales analysis component is declared as passive which means it is created as it is needed for executing services. In order to perform its operations

promptly, the *Sales_analysis* component needs the services of other components. For example, it needs the *theatre* component for getting the total sales value.

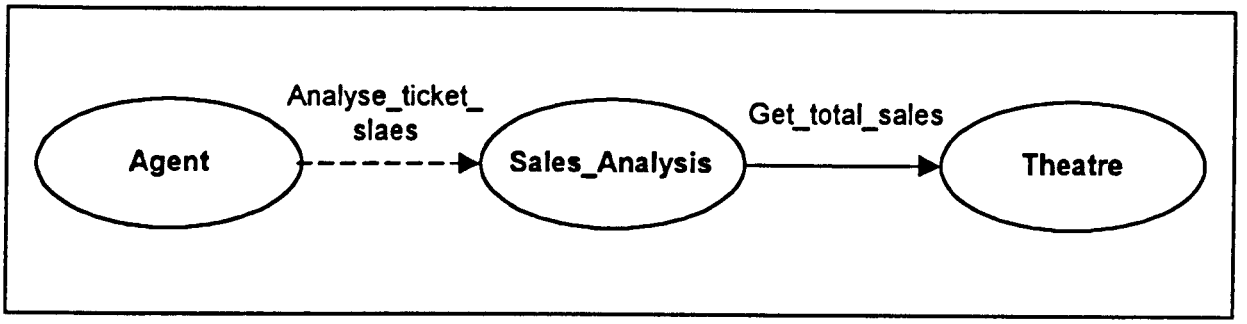

Figure 5-17 Example of Dynamic Relationships

Figure 5-17 shows an example of an architecture schema using dynamic relationships. The schema uses two types of generic models; an *Implicit Creation* and *Client-Server* model. The use of the *Agent* component is very important because the *Sales_analysis* component was declared as a passive component. If the schema had only contained the *Sales_analysis* and the *Theatre* components linked by a message relationship then the schema would have been invalid (because *Sales_analysis* is passive) according to constraint 2 of the client-server model. In other words, we need the *Agent* component to trigger the sales-analysis request in the domain. This is a valid domain constraint which states that sales analysis cannot be carried out independently and needs another active component *(Agent)* for starting the transaction. So the domain constraints are imposed through the choice of the generic model and the component types.

5.6 Summary

In this chapter, we have introduced the generic software architectures which provide a technology for modelling domains through the use of reusable components and generic architectural models. The generic architectural models are used for modelling dependencies between the reusable components and domain constraints by means of standard relationships. The relationships are divided into static and dynamic relationships. In the static relationships, we

model the structural relationships between components, whereas in the dynamic relationships we model how domain-specific behavioural abstractions are synthesised from the reusable components. Three static and four dynamic models are used in the generic software architectures. Each model has a number of constraints that are imposed upon the components and their types that are used in the schema. These constraints determine the way components are linked together in the reference architecture using standard relationships.

We have also illustrated, using an example, how the architectural models are used for describing domain-specific design conceptions. Design alternatives as well as domain-specific decisions are also modelled using generic architectural models. When systems are synthesised, these design conceptions are put into use according to the scope of the developed systems.

Chapter Six

6. DOOR Reuse Process.

6.1 Introduction

DOOR divides the reuse process into two tasks; *domain engineering* and *application engineering*. Both are iterative processes that allow for the evolution of reusable assets in the domain model. *Domain engineering* is concerned with building the domain model assets and *application engineering* deals with identification and retrieval of these assets in the design of new systems [Ramachandran and Al-Yasiri 1994]. Because of the dynamic nature of most domains, where domain requirements and user needs change with time, the two tasks are conducted together every time a new system is implemented. Whereas new systems are synthesised from reusable objects (application engineering), reuse effort is assessed by a set of guidelines and the domain model is updated where needed (domain engineering).

In chapter four we introduced the main parts in the Domain Oriented Object Reuse (DOOR) process. In this chapter, we explain these parts in details. First we discuss the implication of DOOR process on the software development life-cycle. A new modified life-cycle is proposed that is based on domain analysis. In section 6.3, we explain the domain engineering tasks and in section 6.4, we explain the tasks of application engineering. Throughout the process we provide a comprehensive set of guidelines to conduct the tasks involved in the process. The chapter is concluded with a summary of the main points.

6.2 DOOR Software Development Life-cycle.

Figure 4-1 showed the basic elements of the DOOR approach. These elements are incorporated in the process of developing software systems which comprises a number of steps that are applied in sequence. These steps are divided into two phases; Domain Engineering and Application Engineering (see Figure 4-1) and validation procedures for reuse assessment. Figure 6-1 shows the role of the two phases in the development life-cycle for building software systems from domain-oriented components.

The software development process is based on three principles;

1. Incorporating domain analysis and domain-specific artefacts throughout the development life-cycle.
2. Integrating development for reuse (Domain Engineering) and development with reuse (Application Engineering) together.
3. Continuous evolution of the domain model each time a new system is being developed.

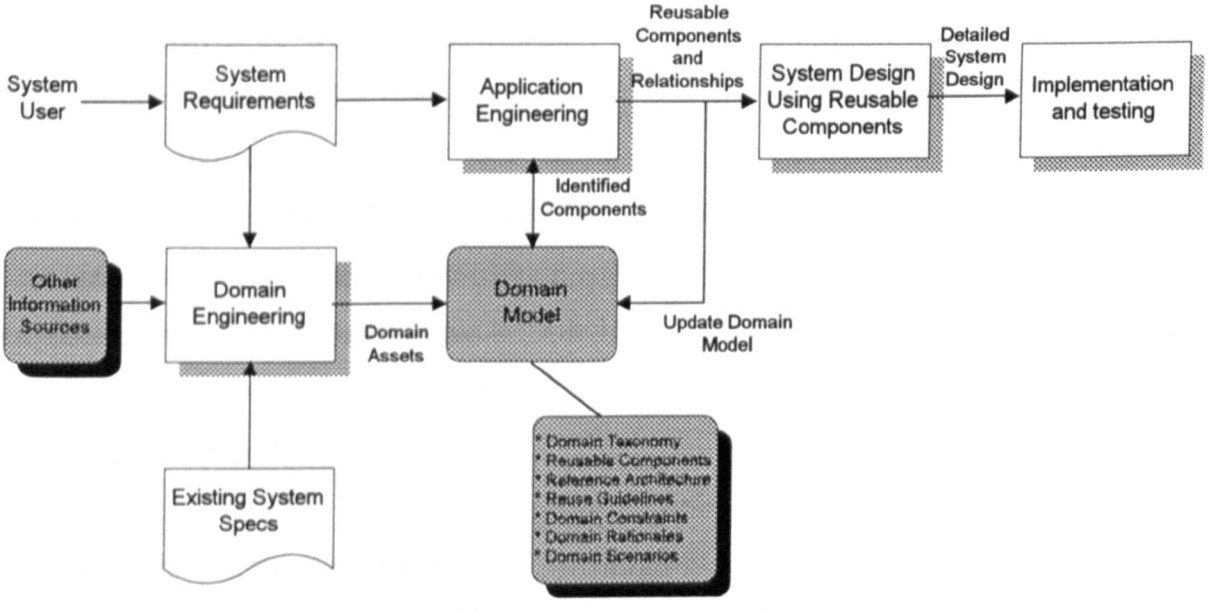

Figure 6-1 DOOR Software Development Life-cycle

The goal is to construct a comprehensive domain model which should provide an accurate description of the domain requirements at all times. Nevertheless, a comprehensive domain model can be constructed only if features of all the systems which belong to that domain are included. Of course it is not easy for a system analyst or even a domain analyst to achieve this goal in the first instance. The way to do that is to use existing systems requirements as a source of information for domain analysis. In addition, the results of analysing new systems are also used to update the domain model. The domain model, on the other hand, provides a knowledge base, which is a repository of domain requirements that may be used when new systems in the domain are analysed. This way, we allow the domain model to evolve as well as ensuring that new systems comply with the domain requirements. The domain model contains information that describes the domain-specific requirements (see Figure 6-1). These are domain taxonomy, reference architecture, reusable components, domain resources and reuse guidelines.

In addition to the system analyst, there is another actor in the process, a **domain engineer** who is responsible for modelling the domain and updating the domain model. The domain engineer's tasks are identifying the common features which are potentially reusable and analysing the properties of the new systems to determine their reusability. If they are already specified in the domain model they can be retrieved and used. If they do not exist, the domain engineer sets design guidelines (in accordance with the domain constraints) for building them in a form suitable for future reuse and the domain model is updated accordingly.

The domain engineer's tasks are outlined in the domain engineering and application engineering steps. The next sections explain these two phases.

6.3 Domain Engineering

Figure 6-2 shows the tasks carried out during the domain engineering phase. Domain engineering is concerned with modelling the domain assets in the

domain model. Domain analysis plays a significant role in the identification of the constraints and rationales that govern the way systems are built within the domain boundary. The various tasks of this phase are described here.

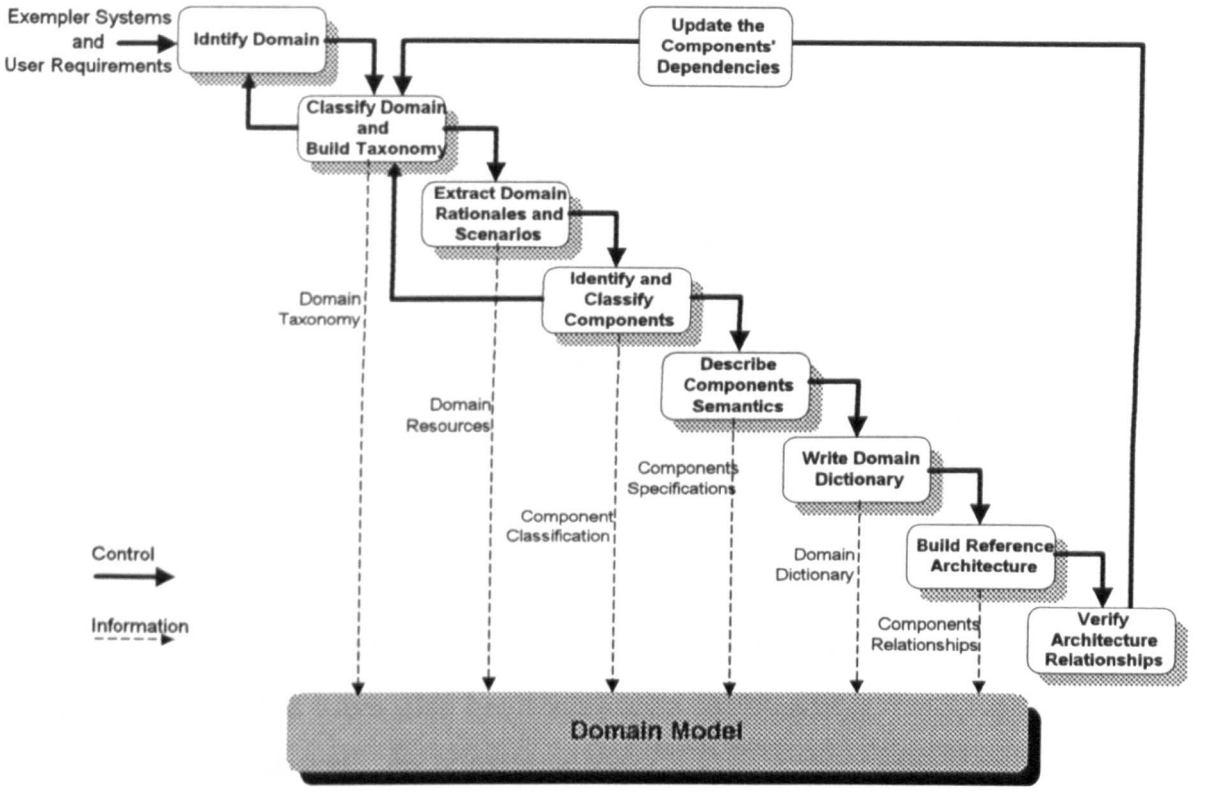

Figure 6-2 Tasks in the Domain Engineering Phase

6.3.1 Domain Identification and Classification

These tasks are shown in Figure 6-2. The first task (domain identification) comprises knowledge acquisition and information collation from different sources of information. The second task is creating the first domain artefact (or modifying the existing one) in the domain model which is the domain taxonomy.

The first task must be executed before the second one since the second requires sufficient information about the domain before it could be conducted. It starts with collecting information about the domain from sources which include technical literature, existing systems, customer surveys, human expertise and current and future requirements. For more information about domain analysis

see chapter two. This task results in good knowledge to identify the domain abstractions and its broad functional requirements. Enough and precise information is crucial to construct a satisfactory taxonomy of the domain. The taxonomy plays a major role in modelling the other assets in the model as well as retrieving them when systems are synthesised. A satisfactory taxonomy means that it reflects the main areas of the domain subject matters and makes the modelling of the other assets easier to achieve. If the domain taxonomy is over-simplified (small number of sub-domains) then its components will be complex and difficult to reuse. It will also be difficult to specify the right form of relationships between the components. If the taxonomy is over-structured (too many sub-domains) then there will be a number of redundant components and relationships. It also makes it difficult to identify the right reuse scope and retrieve the relevant components from the domain model when systems are synthesised.

Structuring the domain taxonomy has been covered in detail in chapter four. A number of guidelines have also been proposed for building the taxonomy which can be used here as well. As shown in Figure 6-2, during building the domain taxonomy we may require additional information from the previous stage and therefore we may need to go back to the first step to collect and analyse more information about the domain. The process may involve a number of consultations between the domain experts and the domain engineer in order to achieve a satisfactory taxonomy.

6.3.2 Modelling Domain Resources

This task involves collecting and extracting two kinds of domain resources (domain rationales and scenarios) from user requirements and existing systems specifications. The third type of resources (domain dictionary) could also be started here, but it may need amendment as the domain model expands, therefore we left it to a later stage.

The reason the rationales and scenarios are set in an early stage of the domain engineering process is that these resources are used in the identification and classification of domain-oriented components which follow this task.

6.3.3 Reusable Components Identification

In this step, we identify the domain-oriented components and classify them according to their scope and group them into kernels which are included in the domain taxonomy. At this stage faceted classification of these components (if applicable) may also be generated to show relations between the components in terms of generalisation-specialisation relationship. The components' classification is based on the components' position on the taxonomy tree. Components may only be related to other components if they fall in each other's reaction space.

The identification of reusable components is a scenario-driven and iterative process. The scenarios are set in the previous stage of the domain engineering tasks. On the other hand, each iteration in the process involves a refinement step in the object specification. The refinement is a responsibility-driven process which uses the domain rationales for guidance.

When components are identified and classified they are organised in kernels and placed on the domain taxonomy tree according to scope (see chapter four for more details). The domain taxonomy may need to be changed according to the components' classification. This is shown in Figure 6-2 by an arrow going back to the task of building the domain taxonomy. More sub-domains may be added to the domain taxonomy or omitted from it according to the availability of components (see guidelines for building domain taxonomies in chapter four). The next step of setting the domain resources may need changing as well. This may only involve re-organising the resources over the taxonomy tree which has been changed.

6.3.3.1 Guidelines for Identifying Reusable Components

General guidelines for identifying objects when systems are analysed already exist [Graham 1991]. Most of these guidelines are based on textual analysis methods where nouns correspond to objects and verbs to methods. Certainly, these guidelines could be used for identifying objects within the domain from user requirements. Objects identified in this way may not be reusable and need to be refined to make them reusable. Guidelines for writing reusable components also exist (see chapter two) which may be applied when such objects are designed and implemented. However, the following guidelines may also be used for identifying reusable components.

Guideline 1

Apply textual analysis guidelines to identify possible objects from user requirements and existing systems specifications.

Guideline 2

Analyse the domain resources set in a previous step and identify explicit responsibilities to each identified object. Such responsibilities must not include access, constructor and destructor or data output functions.

Guideline 3

According to the object responsibilities specified in Guideline 2, identify new objects that could provide services to the objects identified in Guideline 1. Designate responsibilities for each object.

Guideline 4

Use the domain rationales already identified in the domain model to refine the findings of Guidelines 2 and 3. This may involve additional services being added to the objects or some being removed and assigned to different objects.

Guideline 5

Identify objects that have related names, part of a name or adjective and group them together in clusters of objects.

Guideline 6

For each cluster identified in Guideline 5, identify any possible shared services that could be reused by all or sub-groups of the objects.

Guideline 7

New objects should be created for encapsulating the shared services between objects identified in Guideline 5. Attributes for such objects should also be identified and added

to the new objects. The new objects are inherited by the objects identified in Guideline 5, which reuse the features of the new objects.

Guideline 8

For all the objects which are not related (linguistically) by name or part of the name, are analysed for identifying common services between them. If such objects are found then their common features must be identified and encapsulated in abstract classes and their attributes are specified.

Guideline 9

For all objects identified in Guidelines 1-4, look for objects that have a high number of responsibilities. As a general rule the number of responsibilities must not exceed five methods in one object. If any object exceeds the general rule then it should be split in more than one object providing that the resulting objects exhibit high cohesion and low coupling.

Guideline 10

Re-run Guidelines 2-10 looking for more possible common features within the refined objects and carrying out any further refinements until no more refinement is possible.

Guideline 11

For the final version of refined objects achieved in Guideline 7, add a set of auxiliary methods to each object. The list of auxiliary methods comprises a pair of access methods (one for changing value and one for returning the value) and a data output method for each attribute variable.

Guideline 12

For each concrete (non abstract) class identified, add a constructor and a destructor method to its specification.

6.3.4 Component Specifications.

Semantics of each component are defined and documented and added to the components specifications. In our approach, components' semantics are described in terms of what is called the 3-D model of a component as shown in the figure 4-7 and explained in chapter five. The model describes a component in terms of its behaviour, domain scope and its reaction. The behaviour is denoted by the component interface (signatures of all methods of the object); the domain constraints are denoted by the scope of reuse which is part of the object specification in this approach. The reaction space of the component includes other entities whose state (internal values) are expected to change when one of the component's services is invoked. This is denoted in the reaction space of the

components which specifies the domain boundaries for the reuse of a component.

When components (objects) are specified, the following information is included (as shown in Figure 6-3):

1. **Component's Name and Type:**

   The type is determined by the domain-oriented role which is encapsulated by the component's abstraction.

2. **Scope:**

   This is the position of the components the taxonomy tree. In other words, its parent domain name. From the component's scope the reaction space could also be determined.

3. **Behaviour:**

   This is a list of the component services (methods). At this stage, names of the services only are listed here. Complete method signature could be specified later.

4. **Attributes:**

   The last item in the component's specification is a list of its state variables that it is responsible of managing. These are called the component's attributes.

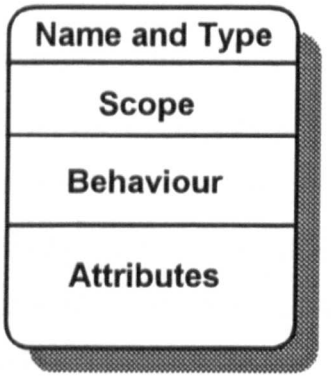

Figure 6-3 Domain-Oriented Component Specifications

6.3.5 Domain Dictionary Compilation

The last of the domain resources to be added to the domain model is the domain dictionary. The dictionary is compiled as item titles and descriptions. The dictionary describes the domain perception of each item and is organised according to scope. If an item is expected to be cited over the whole domain then it should be included in the scope of the main domain. Similarly, if it is only cited in one sub-domain then it should be included in that scope.

There is no restriction upon the information that should be included in the dictionary. It depends on the nature of the domain to decide what information to be included. For example, if the domain is widely known whose items are already familiar then we expect the dictionary to be less comprehensive than a dictionary for a less familiar domain. As a guideline, the following items are recommended to be included in the domain dictionary:

1. Domain-specific acronyms.
2. Domain-specific hardware and devices.
3. Domain-specific procedures and transactions.
4. Real world objects that are found in the domain.
5. Domain-specific events and conditions that happen in the domain.
6. Sub-domains and unity domains in the domain taxonomy.
7. Domain-oriented components whose names are not familiar to the reuser.

6.3.6 Reference Architecture Modelling

Finally, the domain reference architecture and relationships among components are modelled and added to the domain model. Domain architectures are components' relationships modelled using the generic software architectures (see chapter five for details). Each component in the domain model must be linked by at least one architecture schema in the reference architecture. The reference architecture specifies design conceptions in the domain which are used to link components together in order to form valid domain-specific transactions. It uses the domain scenarios as incentives for deriving the domain-specific transactions in the design. On the other hand, the domain rationales are used to impose

domain-specific constraints on the components' relationships. These constraints are modelled in terms of the generic models' constraints and components' type.

6.3.6.1 Guidelines for Building Reference Architectures.

The reference architecture is used to specify dependency among components that are identified in the domain model. The following guidelines are used to identify dependencies in the reference architecture. These are guidelines which we have developed from experience with building reference architectures. Other ways for building the reference architecture are also possible and these guidelines should not stop any one from attempting other ways for building architectures.

Guideline 1

Use the 'is-a' relationship to model classification of components in terms of generalisation-specialisation. Normally, the general type of components are found in a parent domain and the special types are found in sub-domains.

Guideline 2

Use the 'is-a' relationship to model dependency among an abstract component and its concrete children. This type of dependency is used to model a consistent interface between a number of components that have the same interface but vary in the way they are implemented.

Guideline 3

Use the 'is-a' relationship to model situations where the child component represents a restricted condition of the parent component.

Guideline 4

Use the 'is-a' relationship between two or more polymorphic components to model design alternatives. This relationship is mainly used to provide different implementations of a certain service in the domain.

Guideline 5

Use multiple inheritance relationship to merge two functionalities from two distinct objects in order to form another component that combines both functionalities. Such a situation is mainly used to combine a data manager object (an object that is responsible for maintaining data value or values) and a viewer object (an object that is responsible for data display) to form a data manager component with specific viewing functionality. In other words restricting the data display to a certain type.

Guideline 6

Use the association relationship between components that need each other's services in order to perform their own services.

Guideline 7

Use association relationships to model dependencies between one data manager components and a number of viewer objects to imply possible ways for viewing the data values.

Guideline 8

Use the aggregation relationship when components are related in the domain as one or more components are part of another component. As an example a classroom object is linked by aggregation relationship to a school object. If the relationship is meant to model a design case rather than a domain-specific conception then it is more suitable to use the association relationship.

Guideline 9

Use aggregation relationship with 'AND' function to model the case of a number of components which are restricted to happen simultaneously in conjunction with another composite component.

Guideline 10

Use aggregation relationship with 'OR' function to model the case of a number of components are not restricted to happen simultaneously in conjunction with another autonomous composite component. The unrestricted situation means that any component could occur alone or with other components. It also means that all the components may also be omitted since the composite class is autonomous.

Guideline 11

Use aggregation relationship with 'OR' function to model the case where a number of components happening in conjunction with another semi-autonomous composite component to model a situation similar to the one in Guideline 10 except that at least one component must be available.

Guideline 12

Use aggregation relationship with 'EX-OR' function to model the case of a list of components in conjunction with another semi-autonomous composite component. Any component in the list is restricted to happen exclusively without the occurrence of any of the other components.

Guideline 13

Use aggregation relationship with 'EX-OR' function to model the case of a list of components in conjunction with another autonomous composite component. Any component in the list is restricted to happen exclusively without the occurrence of any of the other components. Since the composite component is autonomous then all the components in the list may also be omitted.

Guideline 14

Use a message relationship to model dependency between two components where one component provides a service to the other component for performing a specific service.

Guideline 15

Use a message relationships to model dependency between components that have been one big component and then split into two or more components where services from one components are needed by other components..

Guideline 16

Use an association relationships to model dependency between components that have been one big component and then split into two or more components where services from components are needed by each other. Refrain from modelling the relationship in terms of aggregation because it implies high coupling between the components.

Guideline 17

Use a message relationship with multiple client-server to model service broadcasting from the server component to a list of clients which must all be available for receiving the service in sequence.

Guideline 18

Use a message relationship with concurrent client-server to model service broadcasting from the server component to a list of clients which do not have to access the server in sequence and without synchronisation.

Guideline 19

For every passive component in the domain model find an implicit creation model that must link it to another active component. A good start to look for such relationships is to look among components that are linked to that passive component by an association relationship.

Guideline 20

Use the event-driven model to model a transaction which is described in a scenario associated with an external condition, such as a hardware pre-condition or a signal from an external sensor, or a response from the system user.

Guideline 21

There is a possibility that a situation like the one described in guideline 20 may involve an implicit creation model for activating one or more of the components in the transaction.

Guideline 22

Use the batch process model to model a transaction that involves a repetitive sequence between components and all components in the model are used to change the value of a data entity, the contents of a data file or the of status a hardware device.

Guideline 23

Use a periodic pipeline relationship to model a transaction that involves a batch process that are likely to be repeated several times after the last task in the model is executed.

Guideline 24

Use the event-driven model in association with every periodic pipeline relationship to register the event of terminating the pipeline.

Guideline 25

Use the event-driven model in association with every batch process to register the event of starting the pipeline if the pipeline is not started by a message or a trigger relationship.

Guideline 26

Identify all the components that are linked by an association relationship without any dynamic relationship between them. For each one of such an association, there is a possible evolution in the reference architecture in the future to relate these components by at least one dynamic relationship.

6.3.7 Reference Architecture Validation

The remainder of the domain engineering process is validating the domain architecture. The validation process involves detecting invalid relationships between components and identifying redundant or unsound relationships or components in the architecture. The first type of validation (detecting invalid relationships) is governed by the constraints of generic models which have already been explained in chapter five. Such detection is done instantly and automatically as the relationship is specified with the aid of the supporting tools. The tool checks the type of the components used in the relationship and the constraints imposed by the generic model to validate the application of the relationship. If the constraints allow the combination then the schema is accepted. If the constraints do not allow it then the schema is rejected instantly and the relationship is deleted from the domain model; for more details see chapter seven.

The second type of validation is concerned with improving the reference architecture relationships and enhancing the overall design. This involves identifying any redundant relationships and components, detecting any incomplete transactions and updating the domain taxonomy and its kernel components. Such validation is conducted by following a number of guidelines used for verifying the architecture relationships.

6.3.7.1 Guidelines for Verifying Reference Architectures.

The following guidelines are used for verifying the reference architecture in the domain model. They mainly involve checking a combination of architecture schemas and propose more suitable ways for designing the architecture. These guidelines only propose alternative ways to organise the domain model and do not enforce them. It is left to the domain engineer to decide which alternative they would base their design upon. This is done deliberately in order to give the domain engineer more flexibility in designing the domain model which allows them to lay a specific schema that requires further refinement. Such refinement could be done in the future allowing the domain model to evolve with time and as new ideas emerge from analysing new systems.

Guideline 1

If there are two components linked to each other by one architecture schema such that there are no other schemas that link any of the two components to other components in the domain model, then such a schema may be redundant and the two components could be combined together in one component.

Guideline 2

Consider the situation described in Guideline 1, if the schema is a pipeline relationship, then it is not redundant.

Guideline 3

Consider a component situated in a parent domain and inherited by one or more components situated in one sub-domain of the parent domain. If the component in the parent domain is not linked by any other schema then it may be moved to the kernel of the sub-domain.

Guideline 4

Consider the situation described in Guideline 3, if there is only one component in the sub-domain inheriting from the parent domain component, then the two components may be merged together in one component and placed in the sub-domain kernel.

Guideline 5

Check all schemas that use the batch process model and look for the starting condition of the pipeline. If there is no message or trigger relationships that indicate the start of the relationship, then there should be an event-driven model specified for registering the starting of the pipeline.

Guideline 6

Check all schemas that use periodic batch process models and look for the terminating condition of the pipeline. If there is no event-driven model used for registering when the batch process is terminated, then such a schema is unsound and needs to be checked and modified.

Guideline 7

Consider two components that are situated in two separate sub-domains on the same level of the taxonomy and need to be linked by a client-server model but they cannot be linked because they are not situated in each other's reaction space. The solution to this problem is to introduce a new abstract component which is inherited by the server component. The abstract component must be placed in the parent domain kernel.

Guideline 8

Generate a list of all the components that are linked by an association relationships with no dynamic relationships between them. Such a list may be considered for possible evolution of the domain model in the future.

6.4 Application Engineering.

Figure 6-4 shows the tasks of the application engineering process. The first step in the process is choosing the domain scope to which the desired system belongs and the domain boundaries are identified. One of the major problems with domain-specific reuse has been the identification of the domain boundaries for a certain application. DOOR solves this problem by classifying the domain into a number of subject matters. This makes the identification of the domain scope easier by browsing through the domain taxonomy to find out the subject matter that best fits the new requirements. If you have difficulty finding the right scope for your system the following general guidelines may provide an incentive:

Guidelines for Domain scope identification

1. *Browse through all the unity domains in the taxonomy. These should be found at the lowest level of the taxonomy tree.*
2. *Match the new system requirements with the unity domains on the taxonomy. Locate as many matches as possible; this could be in the form of perfect or approximate matches.*
3. *Make a list of all the unity domain that have matched the new system requirements.*
4. *Follow all branches of the taxonomy tree going up from the unity domains until all branches meet at one parent domain. This parent domain represents the domain scope for that system.*

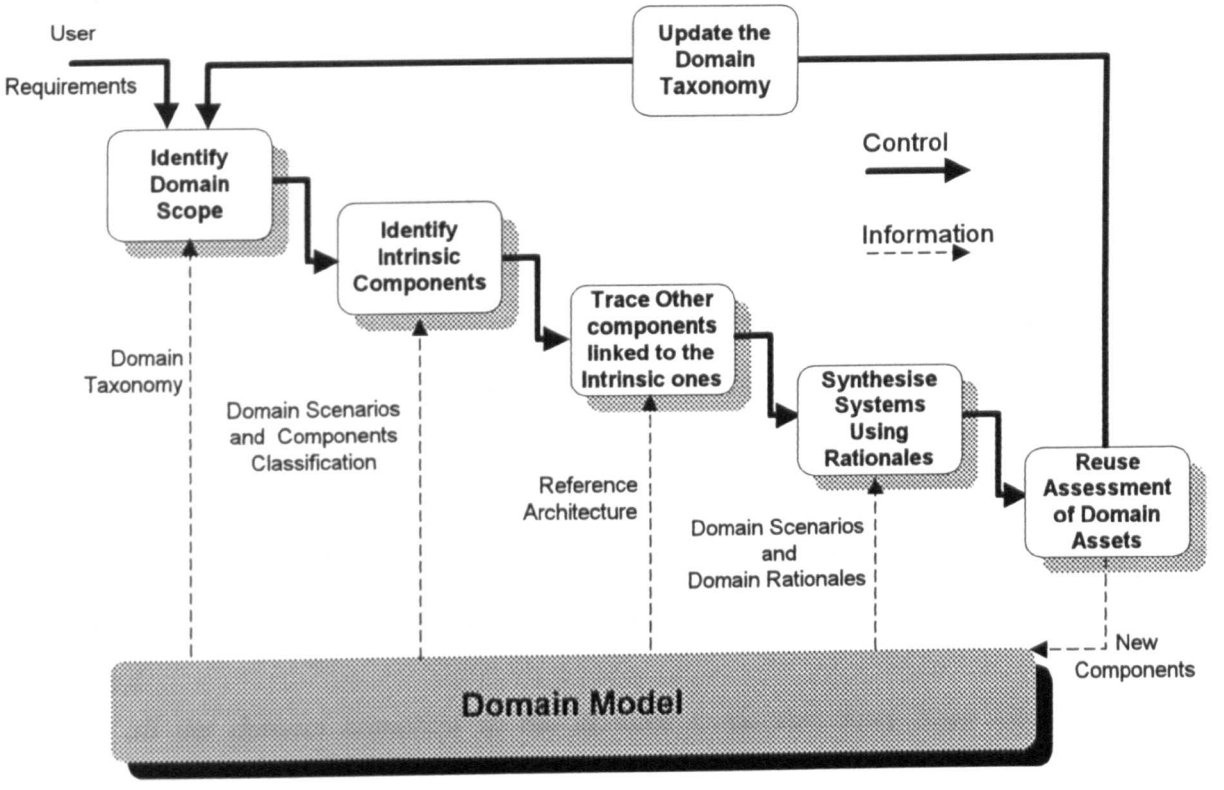

Figure 6-4 Application Engineering Tasks

6.4.1 System Synthesis

Once the scope is defined, a number of reusable objects can be retrieved from the domain kernel. These are the intrinsic and frequently reusable components for that scope. Relevant components that suit the user requirements can be identified using the domain scenarios. So long as these are retrieved the specifications of those components tell us about other components that are part of their reaction space and could be traced using the generic architectures. When all components are retrieved, the system is then synthesised and the domain rationales (non-functional requirements) are taken into consideration. The domain rationales are implemented and enforced on the design by means of the constraints imposed on the generic architectures. Any domain-specific constraints and rationales which could not have been implemented must be taken into consideration when systems are synthesised.

6.4.1.1 Guidelines for Synthesising Systems

When systems are synthesised from reusable components the main concern is how close those components match the requirements of the desired system. There is no guarantee that the existing reusable components provide a perfect match to those requirements. However domain oriented components provide one important incentive to achieve as close a match as possible. This is through the design of purposely built components that are dedicated and cohesive. Broader abstractions are achieved by combining two or more components together that are related by one or more architecture schemas. The process of synthesising systems is scenario driven and inspired by the responsibilities offered by the components retrieved from the domain model. The following guidelines are recommended when systems are synthesised:

Guideline 1

Study all the domain scenarios in the domain scope that have been identified in the domain engineering tasks.

Guideline 2

Match the user requirements to any of the domain scenarios and make a list of the scenarios that provide similar transactions to the ones described in the user requirements.

Guideline 3

All the scenarios identified in Guideline 2 should have already been modelled using a number of reusable components and generic architectural models.

Guideline 4

Identify any intrinsic components and frequently reusable components within the scope of the matching scenarios. Some of these components must be used in the scenarios; make a list of such components.

Guideline 5

Identify all the components that are linked to the components identified in Guideline 4 by static relationships, and fall within the scope of the identified scenarios.

Guideline 6

Identify all the components that are linked by OR or EX-OR aggregation relationships and retrieve the relevant alternatives; use rationales as guidance for choosing the right alternatives.

Guideline 7

Identify the polymorphic components among the retrieved components and trace all components that are linked by 'is-a' relationships to find out the components that provide the relevant implementation according to the user requirements.

Guideline 8

Identify all abstract components among the retrieved components and trace all component that are linked by 'is-a' relationships to find out the concrete components that provide the desired abstraction according to the user requirements.

Guideline 9

For all the components retrieved in Guidelines 1-8, trace all the dynamic relationships that link them to any other components.

Guideline 10

For all the schemas retrieved in Guideline 9, look for the ones that are related to the scenarios retrieved in Guideline 1 and retrieve all components that are involved in these schemas.

Guideline 11

For all passive components retrieved in Guidelines 1-10, retrieve all active components within the relevant scope that are responsible of activating the passive components.

Guideline 12

For all components retrieved in Guidelines 1-10 and are responsible of announcing certain events, identify and retrieve all components that register their services to those events in event-driven models.

6.4.2 Reuse Assessment

The final stage of this process is reuse assessment where the whole reuse effort is evaluated. This takes two forms; the first is assessment of any successful reuse and problems encountered with the existing domain assets; and the second is exploring new possibilities for reuse that may emerge from the current effort. This may be in the form of new components, scenarios, rationale, dictionaries or an adaptation to the existing assets. The results of the assessment could mean that the domain model has to be modified and/or the task is repeated until all possible reuse is explored.

6.4.2.1 Guidelines for Reuse Assessment

The following guidelines are used to assess the reuse effort in the previous stages. These guidelines tell the reuser about the main points to look for when systems are built for assessing reusability of the domain assets and to give the domain engineer new ideas for enhancing the domain model.

Guideline 1

Make a list of all the objects specified in the new system design and a list of all the objects that have been reused without modification from the domain model The following formula gives an indication of direct reuse in this effort:

$$r = \frac{n}{N} \times 100\%$$

where - n: number of reused objects in the system
N: total number of objects in the system
r: system's direct reusability

Guideline 2

Make a list of all the objects that have been reused with modification from the domain model The following formula gives an indication of overall reuse in this effort:

$$R = r + \frac{m}{N} \times 100\%$$

where - m: number of modified reusable objects in the system
R: system's overall reusability

Guideline 3

The higher the value of R the more successful reuse has been achieved. The aim is to achieve at least 50% overall reusability in the system.

Guideline 4

Calculate relative stability of the domain model using the following formula:

$$S = \frac{r}{R} \times 100\%$$

where - S: relative stability of the domain model
The lower the value of S the more the domain model needs evolution.

Guideline 5

Re-analyse the objects identified in Guideline 2 and the modification made to them. Conduct domain engineering tasks to update the domain model accordingly.

Guideline 6

Identify all the objects that are linked by association relationships with no dynamic relationships between them. Analyse the way they were linked in the system and suggest ways to link them in the domain model.

Guideline 7

Identify all the scenarios that have been identified in the system synthesis stage whose components have been modified. Analyse these scenarios and suggest ways to modify them or create new scenarios and add them to the domain model.

Guideline 8

For all the objects that have been reused directly, find out how many times they have been reused in the past and update that number according to this effort. Report the overall number of times those components are reused.

Guideline 9

For all the objects that have been reused with modification, find out how many times they have been modified in the past and update that number according to this effort. Report the overall number of times those components have been modified.

Guideline 10

Identify all the intrinsic and frequently reusable components in the current scope which have not been reused in this system. These components should be recommended to be considered during domain engineering tasks for modification.

6.5 An Example of Domain Modelling

In the last two chapters, we introduced an example of the reservation systems domain. A preliminary taxonomy of the domain was built in chapter four and a list of reusable components has been identified. In chapter five, examples of relationships between some of the components in the domain were demonstrated. In this section, more examples are used to illustrate modelling the domain scenarios and transactions using the generic architectural models.

After classifying the domain and building a domain taxonomy (see chapter four), the next step in the process is modelling the domain resources; dictionaries, scenarios and rationales. These are already modelled and presented in appendix A. Following is identifying reusable components and clustering them according

to scope. Part of this step has been done in the preliminary domain taxonomy. More components could be identified by following the guidelines in section 6.3.3.1. Table 6-1 shows the new components which have been identified with respect to scope. The domain taxonomy is modified accordingly to reflect the final version with the identified components as shown in Figure 6-5.

Table 6-1 Components Scope

Component	Scope	Component	Scope
Customer	Reservation and Inventory SW	Trans_Mgr	Reservation and Inventory SW
Sales_Mgr	Reservation	Purchase_Mgr	Inventory
Stock_Shelf	Inventory	Borrower	Library
Book_Shelf	Library	Library_section	Library
Theatre	Theatre	Theatre_section	Theatre
Seat_Arrangement	Theatre	Config_Mgr	Theatre
Passenger	Airline	Airport	Airline

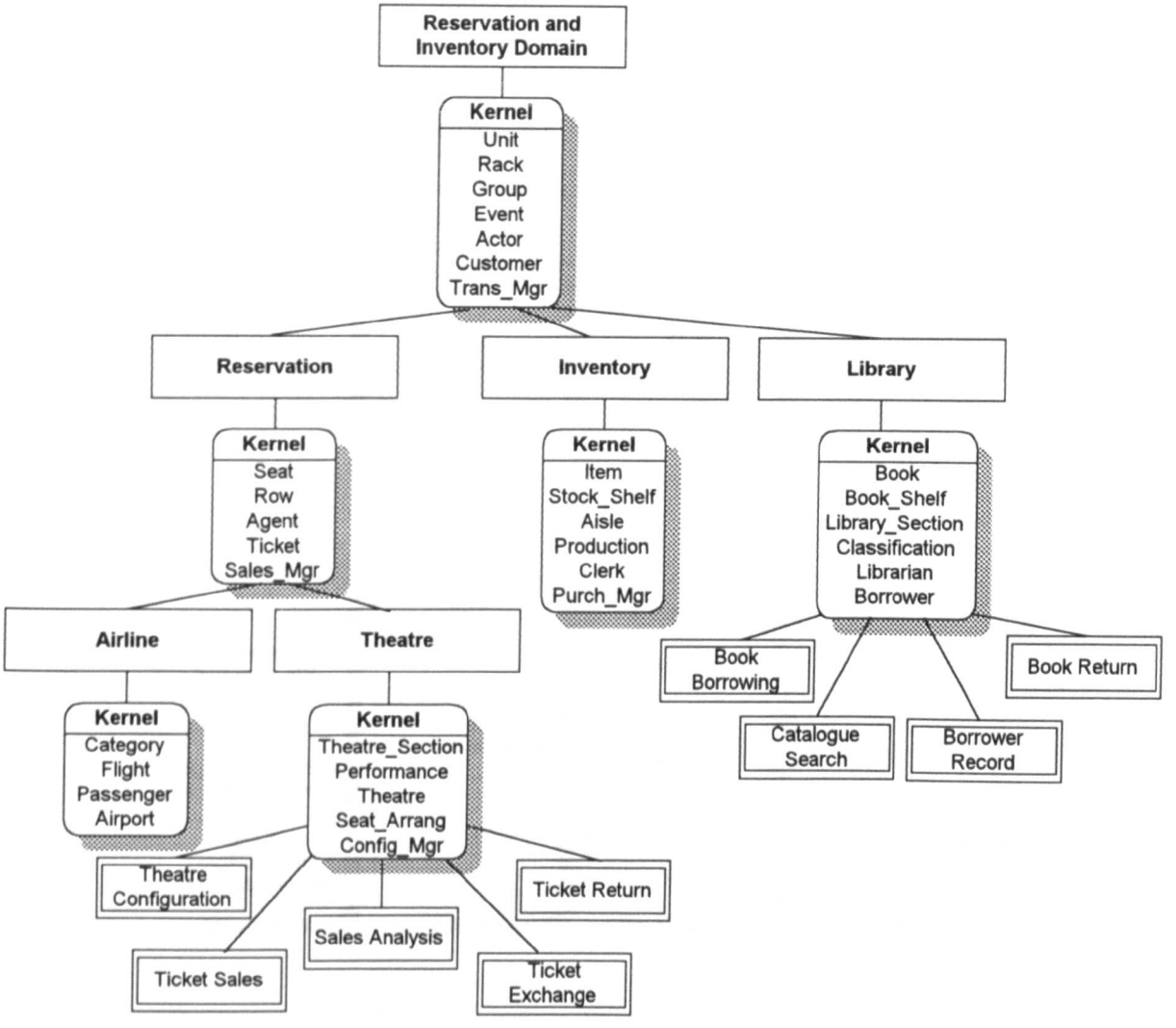

Figure 6-5 The Reservation and Inventory Domain Taxonomy

The next step is specifying the domain-oriented components. Figure 6-6 shows examples of the component specifications in the domain model. As shown in the figure each component is specified with the scope and reaction space in the taxonomy.

The next step is building the reference architecture of the domain which could be divided into static and dynamic relationships. According to the information available to us, the relationships used in the reference architecture are drawn from the theatre domain. However generalised relationships in terms of abstract classes are also included which are relevant to all domains.

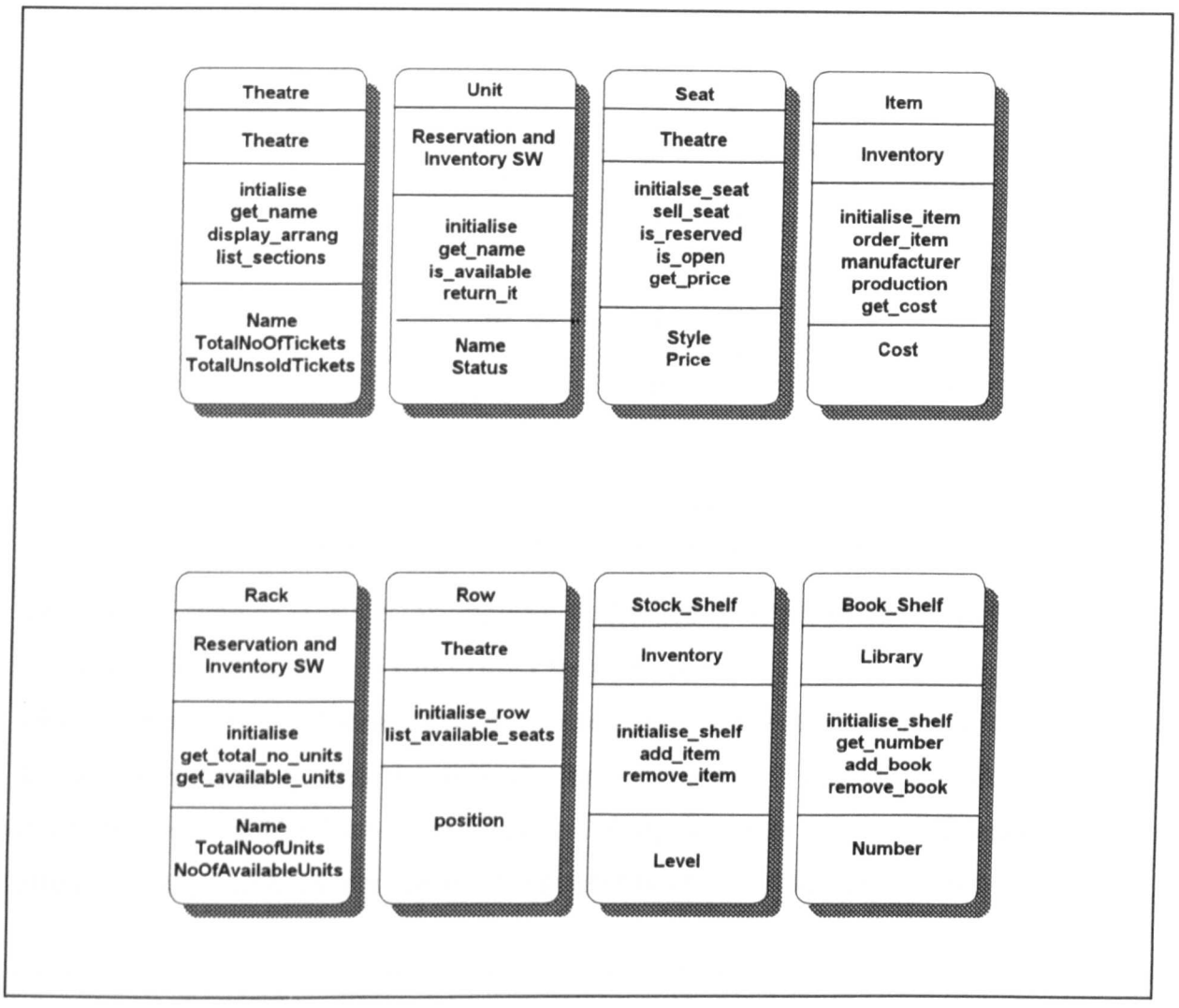

Figure 6-6 Examples of Domain-Oriented Component Specifications

The first type of static relationships are the generalisation-specialisation models in the domain model. These relationships show what features components share and what specific features each one has. Figure 6-7 shows examples of the generalisation-specialisation relationships in this domain. These are useful for two reasons; first they are used to show possible reuse in terms of inheritance and second they are used to show which components behave similarly where design conceptions could be reused when they are specified in terms of abstract classes. The first case is illustrated in Figure 6-7-A and the second is illustrated in Figure 6-7-B and C.

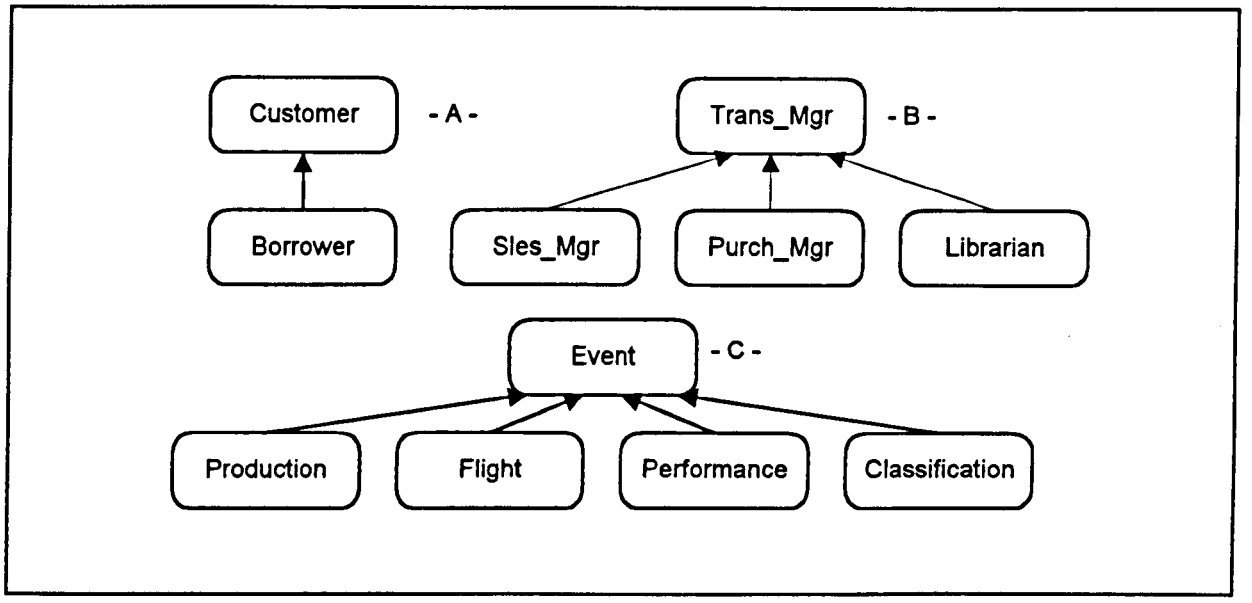

Figure 6-7 Examples of Generalisation-Specialisation Models

The second type of static relationships is the use of aggregation models as shown in Figure 6-8. The figure illustrates two cases; the first is a design conception where a mixture of abstract and concrete components are used (see also section 5-5 for more details), and the second case is relevant to the theatre domain. The interpretation of the first case (Figure 6-8-A) depends on the component types and how they are modelled using the generalisation-specialisation model.

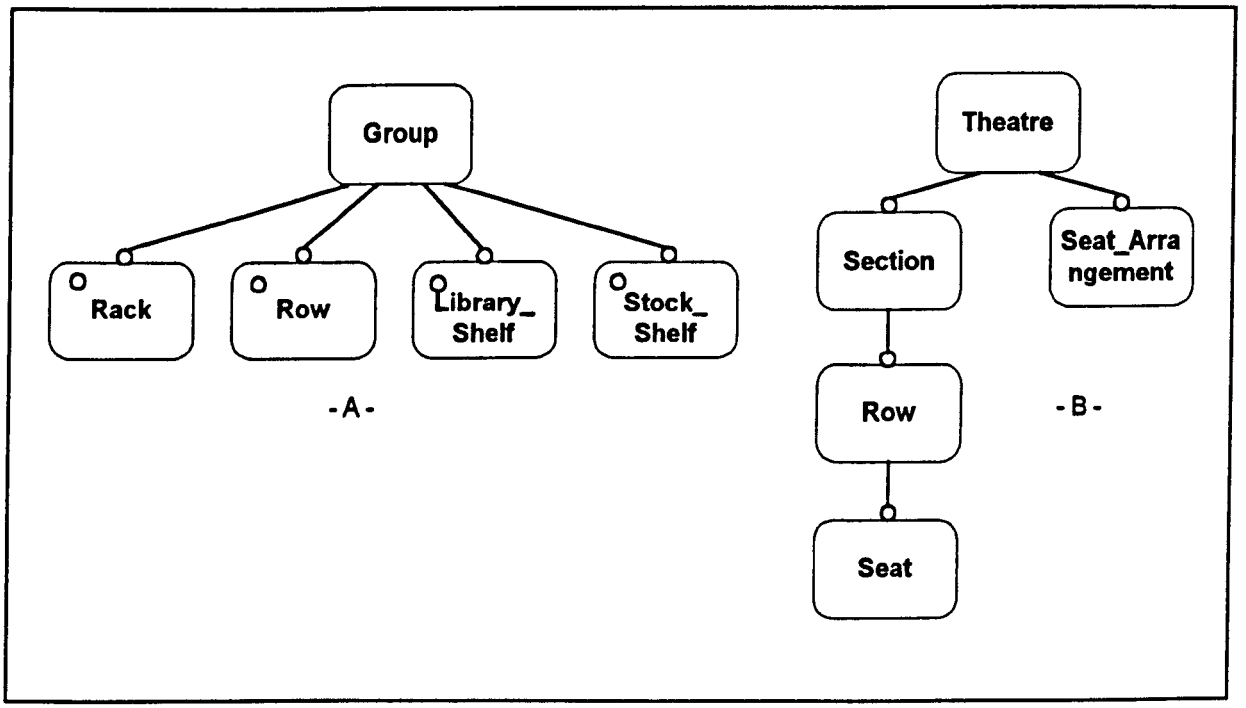

Figure 6-8 Examples of Aggregation Models

The last type of static relationship (association model) is illustrated in Figure 6-9. Again the figure shows a design conception and special cases relevant to the corresponding scopes of the domain taxonomy. For example the case demonstrated in Figure 6-9-B represents a design conception that could be used across different domains where any *Unit* (abstract component) is associated with an *Event* (another abstract component). Both components could be replaced by concrete components according to the relevant scope. Figure 6-9-C illustrates the use of this design conception in the *Library* domain scope whereas Figure 6-9-D shows the *Inventory* domain case.

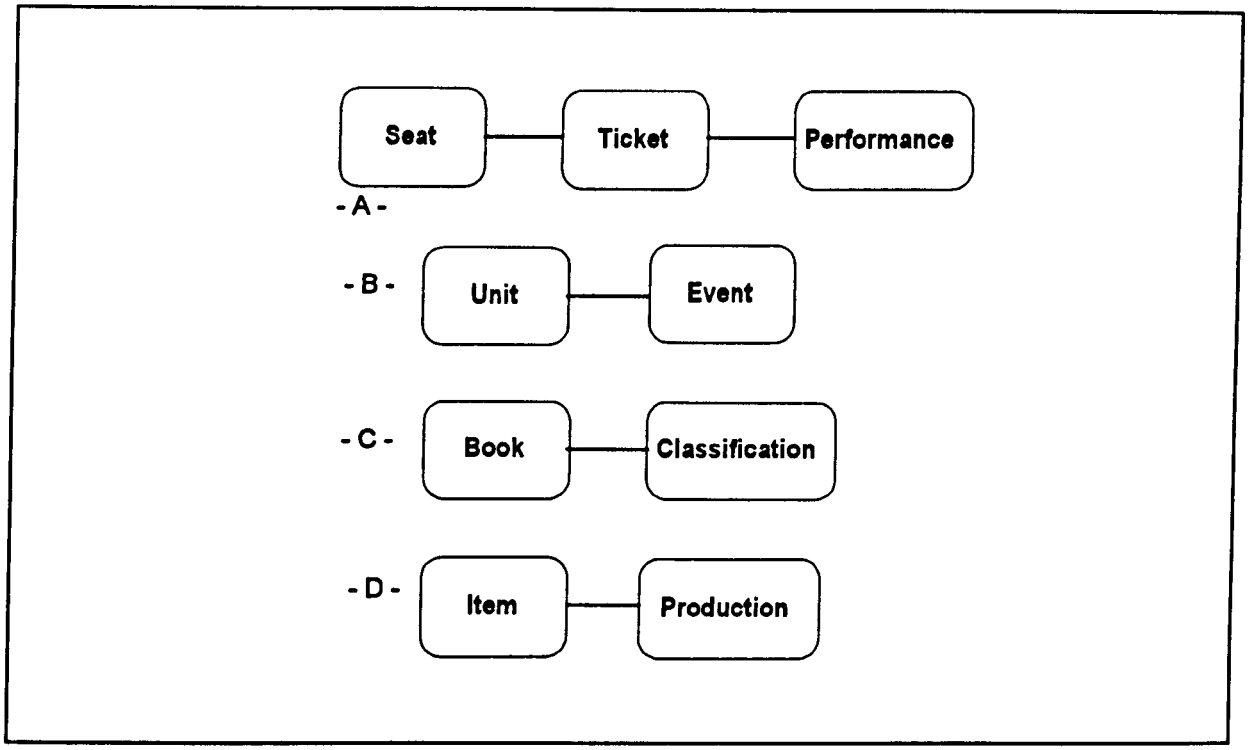

Figure 6-9 Examples of Association Models

In modelling the dynamic relationships, scenarios and transactions in the domain scope must be studied and converted into a number of relationships between components which have been identified from these scenarios. It is very usual that some of the architectural schemas will use other components in the domain model since the responsibilities are shared among different components.

Figure 6-10 shows examples of such relationships. The first example is drawn from the sales analysis scenarios (Figure 6-10-A). In this example the scenario itself is used as a name of one component in the schema. This is because the scenario is modelled as a unity domain and by definition a unity domain is treated as one system or component in the domain. In order to accomplish all the transactions in the scenario, four components are needed. As shown in the figure the *Theatre* component is referenced twice because once it provides a behaviour that is its own responsibility and the other it has to ask one of its agents to provide the required behaviour (see Figure 6-8-B for clarification). In a similar fashion, the theatre configuration and ticket sales scenarios are represented in Figure 6-10-B and C respectively.

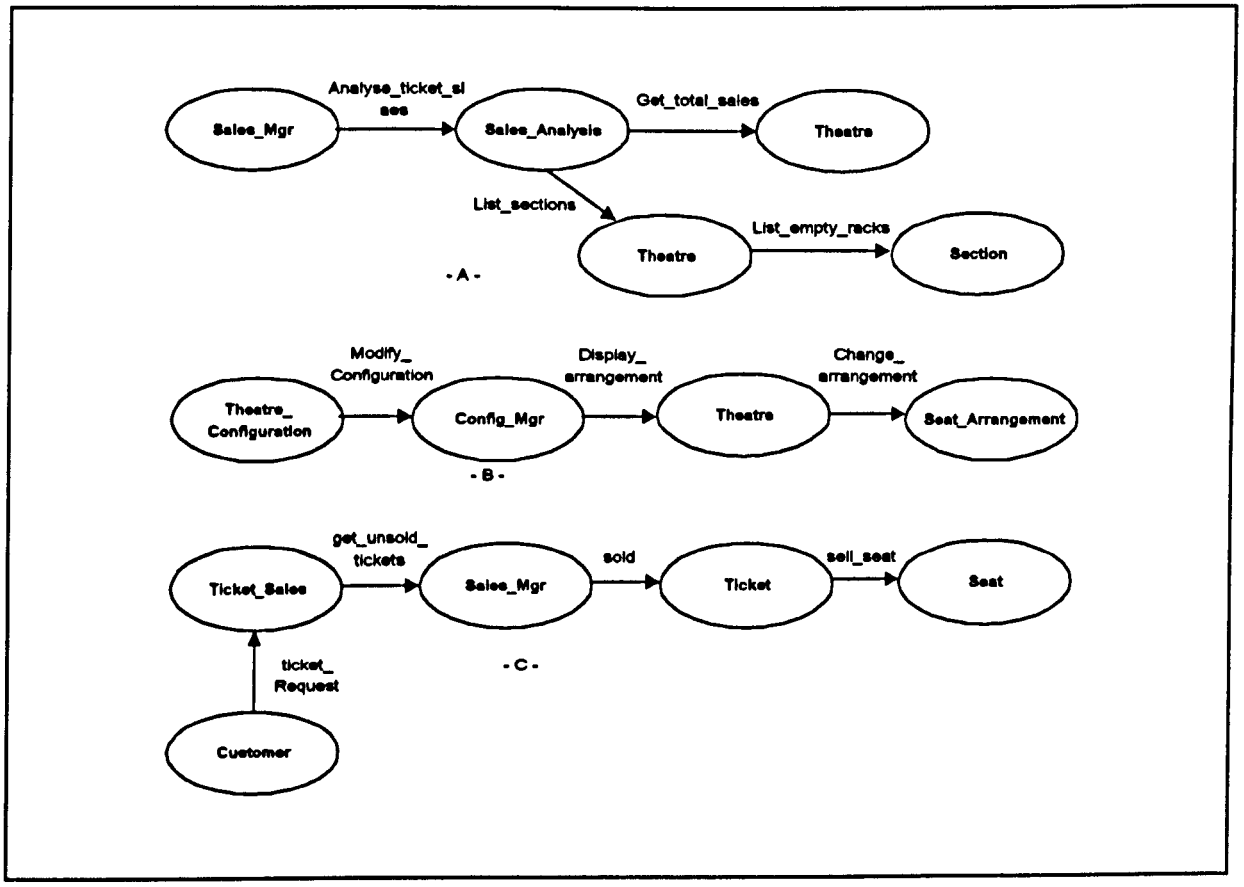

Figure 6-10 Examples of Dynamic Relationships

When systems are synthesised within a specific domain scope the first thing that we need to determine is scope. This could be done by following the guidelines in section 6.4. In this example, identifying the domain scope is straightforward

since the sub-domains are organised in a way that are self-explanatory. This makes the search for reusable components easier. Once the scope is identified (a theatre domain for instance), a list of intrinsic and frequently reusable components could be retrieved from the taxonomy tree (refer to Figure 6-5). These are:

Theatre, Theatre_section, Performance, Seat_arrangement, Config_Mgr, Seat, Row, Agent, Ticket, Sales_Mgr.

The rest of the components that we could identify are bounded components. These are components that are only used for a particular transaction (unity domain) within the theatre scope such as *Theatre_Configuration* or *Ticket_Sales.* If we are building a system to help in the theatre configuration, for instance, then the *Theatre_Configuration* component is retrieved as well .

Other components are retrieved from the domain model by following the architectural schema that use the above components. For example, the *Performance* component inherits from the *Event* component within the scope of the main domain and therefore its behaviour is reused in the theatre scope. The theatre domain reference architecture also tells us how the relationship between *Theatre, Section, Row, Seat* and *Seat_Arrangement* components as shown in Figure 6-8. These architectural schemas represent domain specific architectures that are expected to be used in any system built within the theatre scope.

More components or design conceptions are retrieved from the dynamic relationships in the domain model such as the one shown in Figure 6-10-B. Thus, domain-specific generic architectures provide design architectures for families of systems that could be instantiated when systems are built. The other observation is that, as bounded components are used in an architecture schema the design becomes more specific to a certain situation or system within the domain (as the case with Figure 6-10). This proves that unity domains represent the lowest form of reuse with the domain taxonomy, however they are important for modelling the application of reusable components in the domain as well as variation among domain behavioural abstractions.

6.6 Summary

In this chapter, we have introduced the reuse process in DOOR. This process was divided into two phases; *Domain Engineering* and *Application Engineering* which are executed simultaneously and recursively. The nature of the reuse process is organised in a way that supports the evolution of the domain model and its artefacts. This is achieved by taking into consideration the domain-specific features and requirements that may be identified when new systems are analysed and developed. Furthermore, when the domain model assets are reused, the process is assessed to verify the domain-specific design as well as identifying possible means for updating the domain model in terms of adapting the domain architectures or introducing new artefacts.

The process is supported by a comprehensive set of guidelines for guiding the domain engineer in the development and adaptation of the domain model. The guidelines introduced in this chapter covered the following aspects of the reuse process:

1. Building domain taxonomies
2. Reusable components identification
3. Building reference architectures
4. Reference architecture verification
5. Domain scope identification
6. System synthesis
7. Reuse assessment

In the last section of this chapter, we introduced a number of examples for modelling reference architectures and synthesising systems within the domain of reservation and inventory systems. The examples illustrated the use of generic architectures in modelling design conceptions and components' relationships within the domain and how components are identified and retrieved from the domain model.

The process could also be conducted automatically using a supporting tool which helps build domain taxonomies, architectures and verifying them. This tool is introduced in the next chapter.

Chapter Seven

7. Description of DOOR Tools for Storing and Retrieving Domain Assets

7.1 Introduction

In this chapter, the supporting tools for the DOOR process are described. The tools are used for both building the domain knowledge base and retrieving the reusable artefacts when systems are synthesised. The discussion starts with an overview look at the main features and functionality of the tools as an integrated environment. In section 7.3, the tool support for modelling and representation of the domain assets is outlined. Separate tool for specifying each domain artefact is used; these are *taxonomy editor, architecture editor* and *object specifier.* A separate integrated tool is used for modelling the domain resources; this is the *resources editor.* In section 7.4, the automatic retrieval of the domain assets is discussed.

Section 7.5 introduces a real world case study of a typical domain with modelling the domain resources and artefacts. The chapter is concluded with a summary of the main points.

7.2 Overview and Main Features of the Tools

The tool is an integrated environment for organising and presenting domain knowledge and reusable assets. Figure 7-1 shows the architecture of the tool. It shows how the tool supports both design for reuse and design

with reuse. Within design for reuse, the DOOR assets are specified and added to the knowledge base using the taxonomy editor, architecture editor, object specifier and resources editor. When systems are synthesised (design with reuse), the reusable artefacts are used to identify and retrieve components that are necessary for system integration using the system synthesiser.

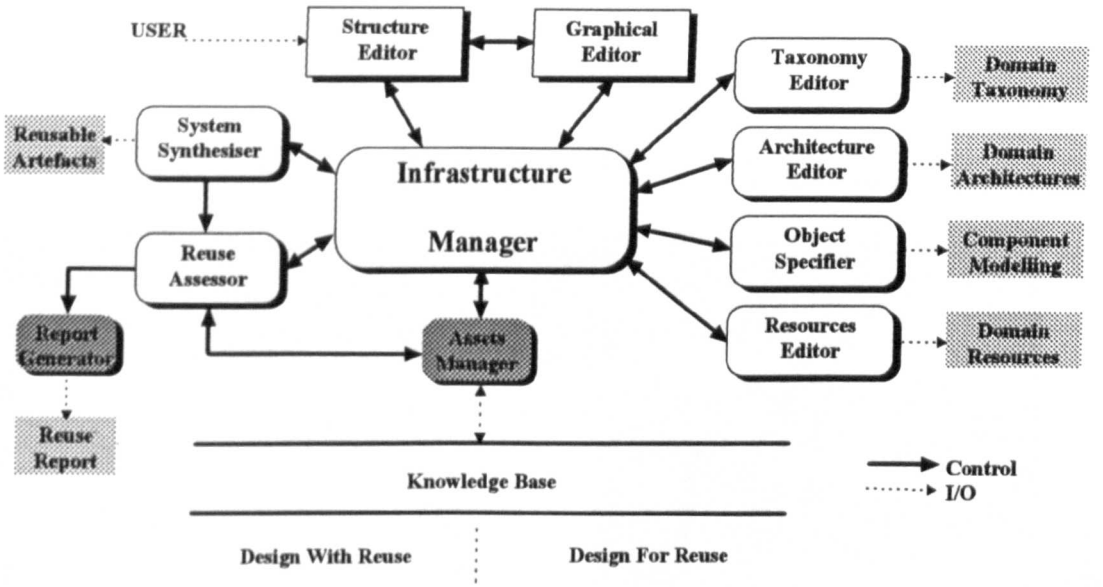

Figure 7-1 Context Diagram of DOOR Tools

The taxonomy and architecture editors allow the user to specify the domain taxonomy and reference architectures graphically, whereas the object specifier and resources editor are used to specify reusable components and domain resources textually. The structure editor and the graphical editors are used by other parts of the tool for displaying output and communicating with the user. On the other side, the reuse assessor is used for checking validity of the overall domain architecture design. The infrastructure manager controls the flow of information between the different parts of the tool as well as managing dependencies among the domain assets.

The DOOR tools are designed using object-oriented techniques. All different tools as well as domain assets are modelled as objects and stored

in the knowledge base using persistent objects. The assets manager manages the flow of data between the knowledge base and the other parts of the systems. It uses two types of data files for this purpose; one for storing domain taxonomies and architectures using persistent objects and the other is for storing HTML (Hyper Text Markup Language) source code for future retrieval. The latter is mainly used for storing domain resources. Figure 7-2 shows the DOOR environment and its integrated tools.

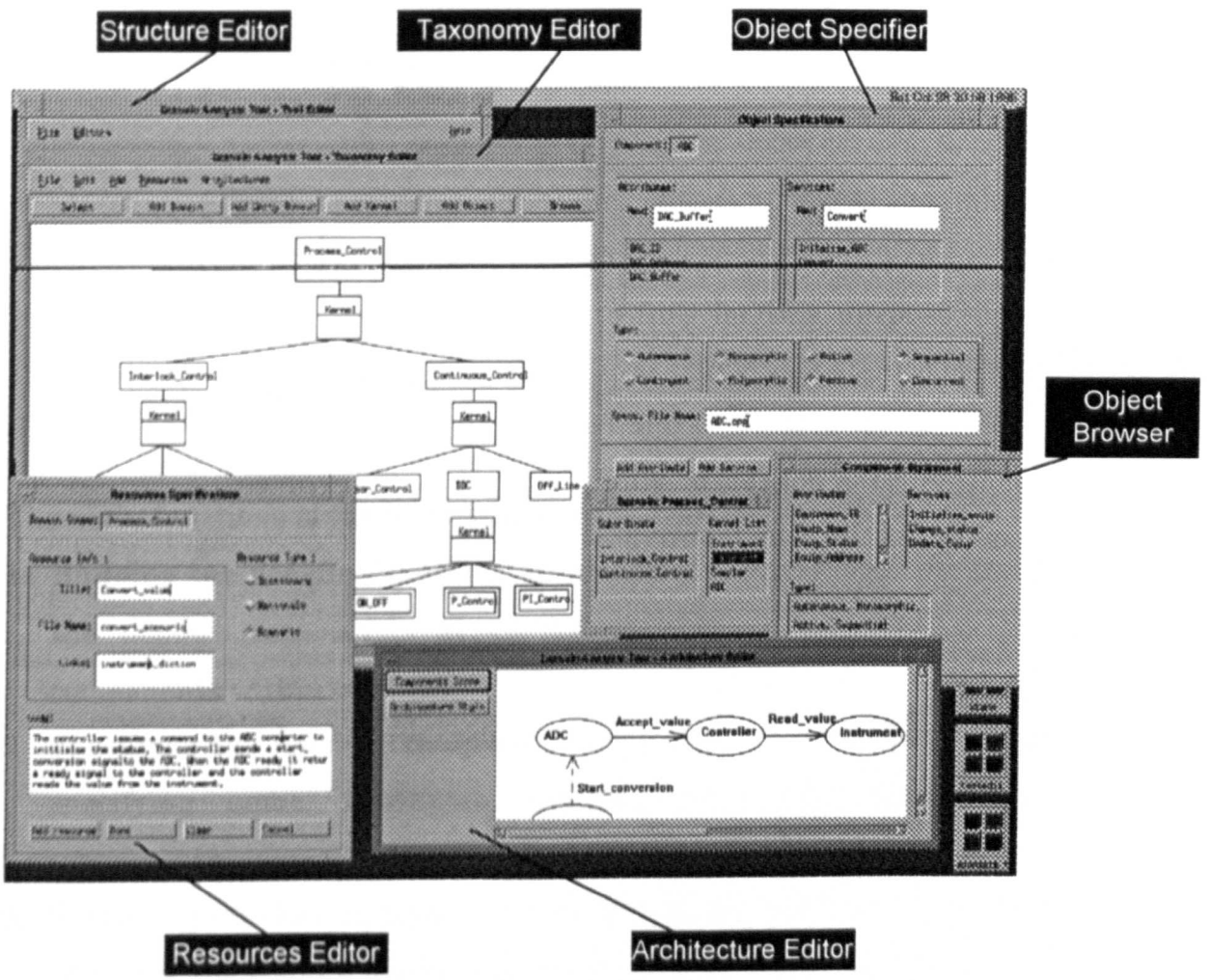

Figure 7-2 DOOR Tools Environment

Figure 7-3 shows the object diagram for the system. All parts of the domain taxonomy (including reusable objects) are modelled using an abstract class called *'Abstract Domain'* which enables the tool to

manipulate the domain model artefacts effectively. Thus all tools recognise only one abstract type of entities whose specific types will be determined within the relevant tools. The *Abstract Domain* classification and inheritance diagram are discussed later in this chapter.

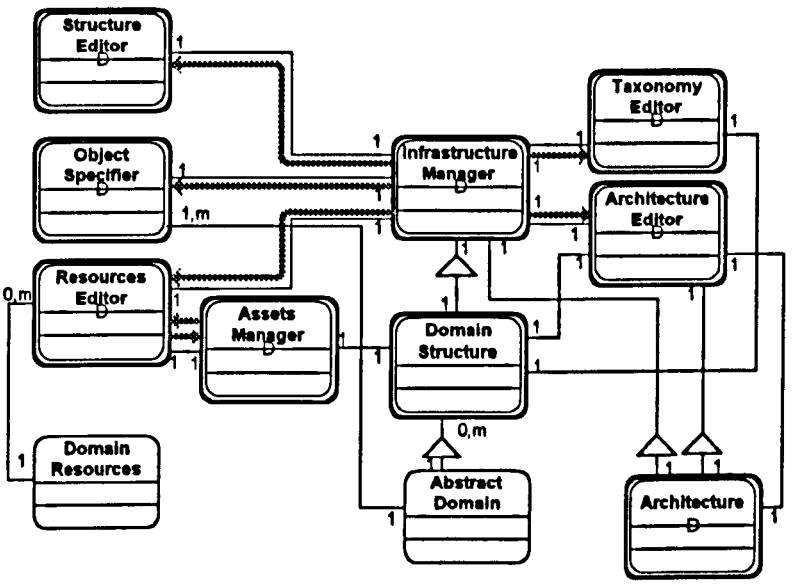

Figure 7-3 Object Diagram of DOOR Tools

The main functions of the tool could be summarised as follows:

1. Building the domain taxonomy graphically. This includes structuring the domain into levels of sub-domains and identifying reusable components within each scope.
2. Specifying the reusable components according to scope. This includes, specifying name, attributes, methods, type and scope of the components.
3. Specifying the domain resources and generating HTML source code for accessing the domain model using the WWW (World Wide Web).
4. Building the domain reference architecture using reusable components and relationships as specified by the generic architectural models. This also includes checking the validity of each schema within the architecture.

5. Allowing the user to browse through the domain model for identifying and retrieving domain assets from the domain model. This is used either by using the tools in the DOOR environment or through an HTML browser.
6. Checking the validity of the overall design of the reference architecture and reporting any design errors or redundant schemas in the design.

7.3 Automatic Modelling of Domain Assets

Reusable assets have been introduced in chapter four of this thesis and their object model is shown in Figure 7-4. The domain assets are either resources or artefacts.

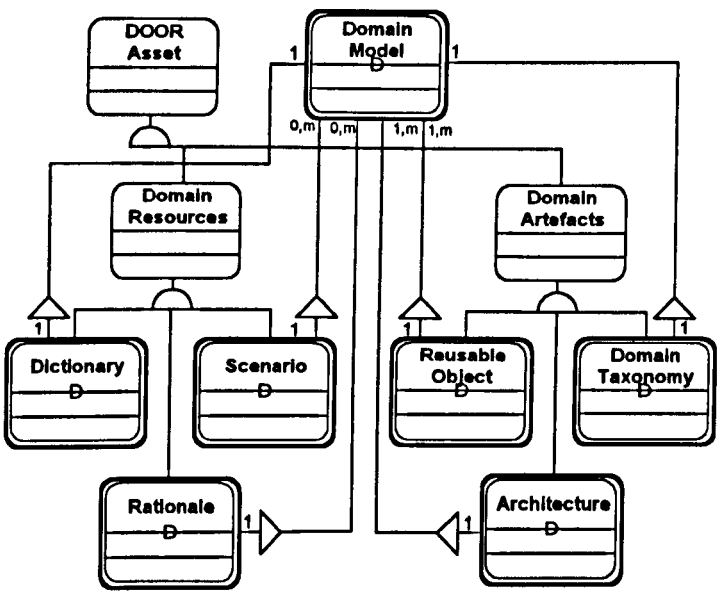

Figure 7-4 DOOR Reusable Assets Classification

The domain resources are information items which are collected from different domain analysis sources of information and organised in Dictionary, Scenarios and Rationales. These are stored in the domain model (domain knowledge base) as HTML source code and accessed using an HTML browser. The artefacts are reusable work products which are developed by the domain engineer and added to the domain model. These are the domain taxonomy, reusable components and reference architecture. Three of the DOOR tools are used for specifying the domain artefacts which are the taxonomy editor, architecture editor and object

specifier. As shown in Figure 7-4, the domain model is connected by a whole-part relationship with the domain assets but with different cardinalities. For instance, there are only one *Architecture* and one *Taxonomy* in the domain model whereas you could find a number of reusable objects in the same model.

7.3.1 The Taxonomy Editor

The taxonomy editor is used for building domain taxonomies graphically. Figure 7-5 shows the components of a domain taxonomy as designed within the tool.

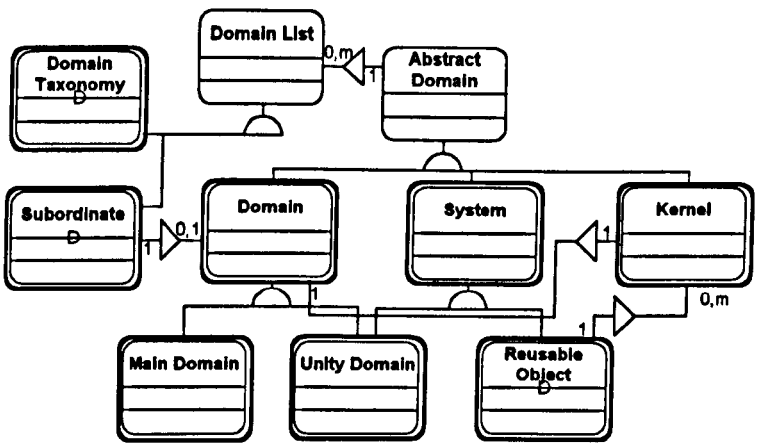

Figure 7-5 Domain Taxonomy Components

As has been stated in section 7.2, an abstract class (*Abstract Domain)* is used to model all parts of the domain taxonomy, which itself is a sub-class of another abstract class (*Domain List*). The use of these abstract classes allows the taxonomy editor to structure domains as trees of sub-domains and kernels of reusable objects as was described in chapter four. The domain taxonomy and the subordinate are modelled as a list of abstract domains. The domain taxonomy contains domains, unity domains, kernels and objects, whereas the subordinate contains domains and unity domains. Each domain contains one subordinate and one kernel and a kernel contains zero or many objects.

Figure 7-2 shows the user interface of the entire environment. The taxonomy editor is shown in the figure as a graphical editor which is designed as a user friendly and easy to use application. It uses a drag-and-drop approach to building different components of the taxonomy. It also instantly checks the validity of any action taken when taxonomies are built. For instance, it does not allow you to add two kernels to a domain or a sub-domain in the taxonomy as this is not permissible within DOOR. using object modelling.

7.3.2 Architecture Editor

As stated in chapter five, a domain architecture contains a number of architecture schemas. Each schema has two or more components (reusable objects) which are related using one or more generic architectural models. Figure 7-6 shows the generic architectures as modelled in the system.

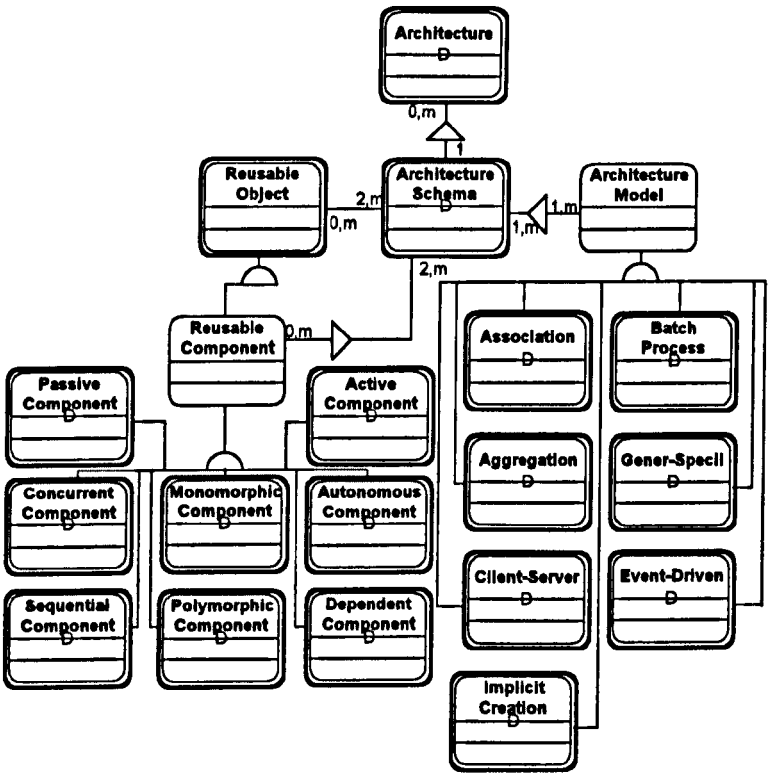

Figure 7-6 Generic Architectures Object Model

DOOR uses seven generic architectural models for modelling the architecture schemas, which are client-server, aggregation, generalisation-specialisation, association, batch process, implicit creation and event-driven models. The reusable components are classified into eight different types as shown in Figure 7-6. Each type is associated with a number of constraints regarding which model could be used for connecting with other components. These constraints are used for checking the validity of the architecture design. For more details about generic architectures, architectural models, reusable components and constraints refer to chapter five.

As is the case with the taxonomy editor, the architecture editor uses a graphical editor for building reference architectures and adding them to the domain model using a user friendly interface (see Figure 7-2)

7.3.3 Object Specifier

This tool is used for specifying reusable objects. Figure 7-2 shows the object specification tool as part of DOOR tools. Objects are specified in terms of their attributes, services and type. The object scope and reaction space are determined through the domain taxonomy and therefore the user need not worry about the scope and reaction space. The object specifier generates a list of attributes and services for that object and adds them to the domain model with the type, scope and reaction space. It also generates an HTML source file and stores it in the knowledge base which can be accessed by an HTML browser.

7.3.4 The Resources Editor

Similar to the object specifier, the resources editor provides a tool for specifying different resources and storing them in the domain model. It too generates an HTML file that could be accessed through the WWW using a suitable HTML browser. Figure 7-2 shows the resources editor (as part of DOOR tools) which is used for specifying the three types of the

domain resources (dictionary, scenarios and rationales). The only way to retrieve these resources during design with reuse is through the WWW. The tool also allows different resources to be linked together so that they could be accessed through other resources. This is done in the HTML files by providing links from one resources file to another

7.4 Domain Assets Retrieval

In this work, a knowledge-based approach to software libraries is adopted. The approach is supported by information representation method which is based on the scope of reuse and behaviour of the components in the knowledge base. So far, in this thesis, the issues concerning the representation of the knowledge base items have been discussed. One of the main obstacles in building component libraries is how to represent the components' functionality in the domain [Maarek, 1993]. In previous chapters, modelling of components' behaviour has been introduced which is used as a basis for representing the components and relationships among them in the knowledge base. In this section, we concentrate on the use of this model for retrieving reusable components from the knowledge base.

An enumerated approach to domain classification is used for organising the domain assets in the knowledge base. In an enumerated classification scheme, the domain is broken into mutually exclusive classes [Frakes and Gandel, 1990]. Domains, in our approach, are classified into a number of sub-domains that represent the scope for reuse and retrieval indexing mechanism. Thus mapping components retrieval to their application in the domain which provides a context for the component's application at the same time it is retrieved.

Figure 7-7 shows how the domain assets are stored and accessed in the domain model (knowledge base). As shown, the knowledge base is central

to the approach, however the domain taxonomy is both a representation and indexing mechanism for storing and retrieving the domain assets.

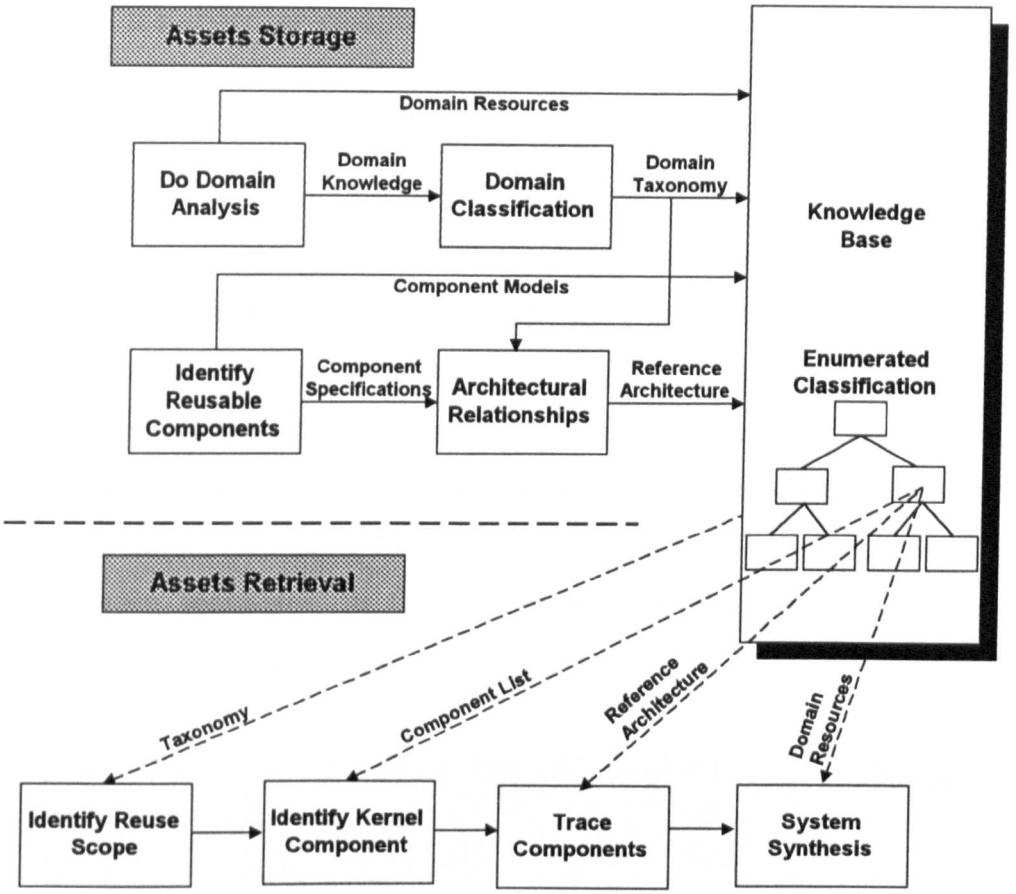

Figure 7-7 Domains Assets Retrieval

DOOR tools support design with reuse by allowing the user to browse the domain model for finding and retrieving the relevant assets. The key point to retrieving reusable assets is the identification of the relevant domain scope. DOOR tackles reuse from the scope point of view. Once the scope is identified, the tools will be able to locate the relevant assets for that scope. This is very useful since these assets were designed to be used within that scope in the first place. On the other hand, the search for relevant components is more focused to the domain abstraction and its applications. In contrast, other component classification schemes, such as the faceted classifications scheme [Prieto-Díaz 1987], suggest schemes for classification that emphasise the functionality of the component rather

than its application. This may cause the reuser to be unsure of whether the component is the right one for the application. In DOOR, the component's scope and relationships with other components describe how and where a particular component could be reused within a domain or a sub-domain.

After identifying the domain scope, its kernel components are retrieved. These are easily found by browsing the kernel contents in the tool. Figure 7-8 shows the domain browsers where the user could skim through the domain subordinate and kernel or to go back to the parent domain. When a component is found its specifications could be displayed and retrieved. Components in the kernel could also be used to trace other components in the domain model and their relationships which may be used, with the aid of the domain scenarios, in the design and implementation of the new synthesised system (see Figure 7-7).

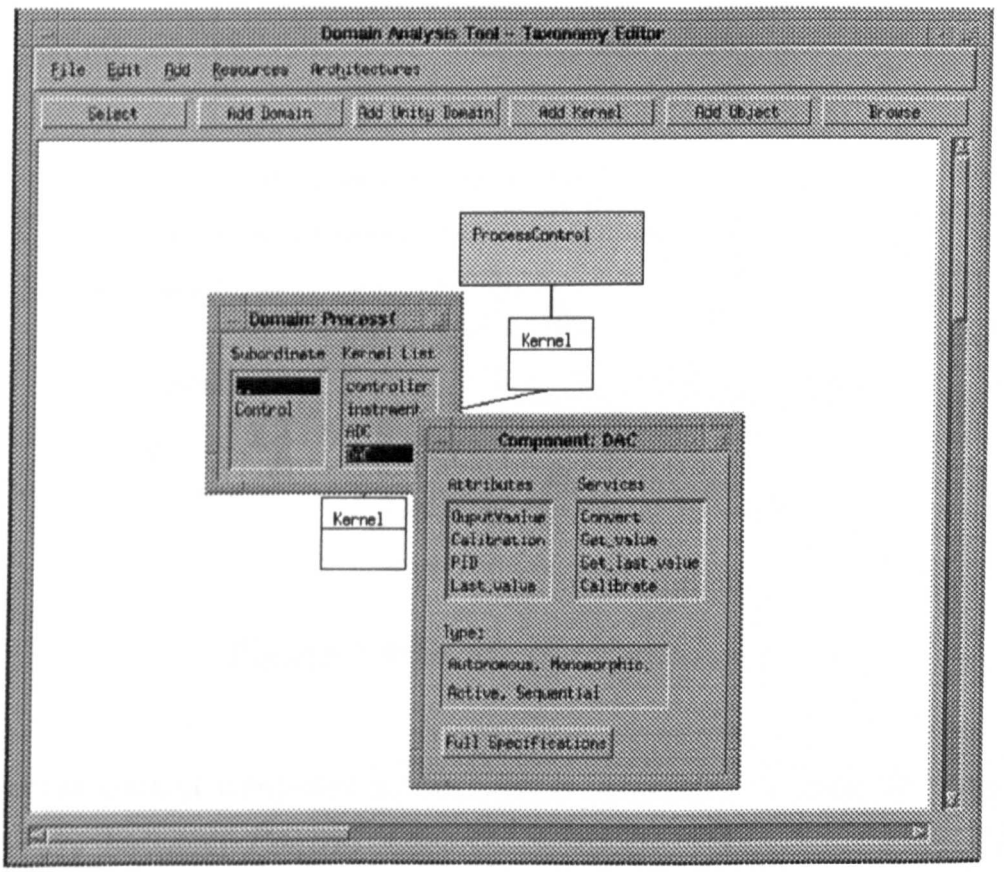

Figure 7-8 Browsing the Domain Model Assets

There is also the possibility of the whole reference architecture being examined and traced looking for relevant design schemas and components to be used in a specific design. Domain resources could be accessed using the WWW and HTML files. In the next section, a case study is used to show how different assets are modelled and retrieved using this tool.

7.5 Case Study (The Process Control Domain)

The domain of process control is a good example of hierarchical domains that could be broken down into a number of sub-domains. Figure 7-9 shows a sketch of the role of computer in the process control domain. The variety of tasks that a process control software is responsible for makes this domain an interesting and challenging domain to analyse.

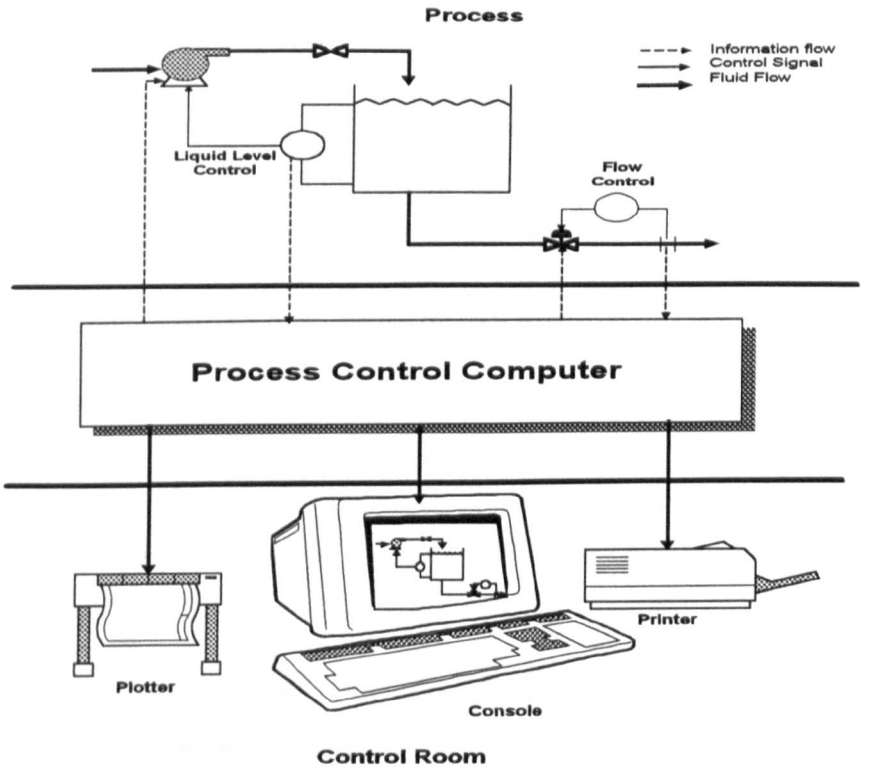

Figure 7-9 Process Control Domain

A process control computer is usually responsible of a number of tasks that vary between *Control Algorithms, Control Room Activities, Data Acquisition* and *Product Quality Control.* The main aim of the process

control software is regulating and maintaining the controlled variables (in the process) within certain limits to ensure the quality of the product.

Previous research in the domain of process control has shown the challenge of analysing this domain [Leveson 1990; Matsumoto 1993; Halang and Kramer 1994; Leveson, et.al. 1994; Pirklbauer, et. al. 1994]. However most of these research attempts have concentrated on one aspect of the process control software. They also failed to show how different parts of the domain are related. In this case study, the whole of the process control domain is modelled as the main domain which comprises a number of sub-domains as shown in Figure 7-10

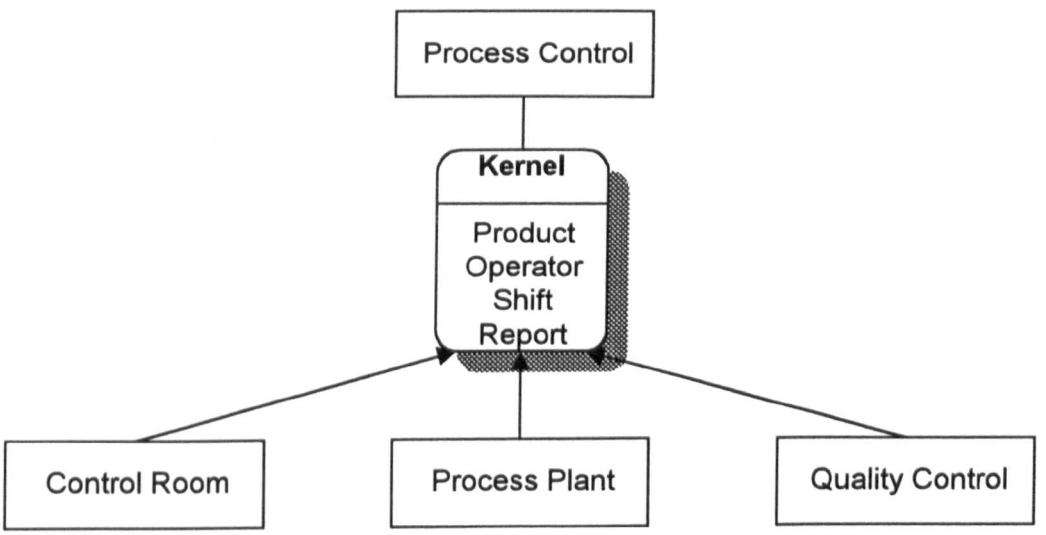

Figure 7-10 Process Control Sub-domains

In Figure 7-10, the process control domain is divided into three sub-domains which share some common objects from the main domain kernel. All relationships between the sub-domains are made through these objects. The process plant sub-domain represents the domain where software components that are responsible for controlling and monitoring the process behaviour are modelled. Therefore this abstraction comprises three closely related sub-domains which are *Testing, Control* and *Data* Acquisition. Appendix-B shows a detailed classification of the entire

domain of process control software. In this case study, the process plant sub-domain is modelled in more detail and in particular the control abstraction is emphasised. A number of preliminary reusable components are identified and classified in Appendix-B. These components are distributed along the three sub-domains, however no attempt was made to define their scope within the domain taxonomy. From the first stages in analysing this domain, it is obvious that there is an overwhelming amount of information to analyse; this is a common feature among all real-world domains. For instance, the classification of the reusable components in Appendix-B is not enough to understand their functionality or how they are applied. There is a need for more information about how these components are related and how they are used for synthesising new systems. Applying the DOOR approach will clarify a number of issues that the reuser faces when trying to reuse these components. DOOR allows the results of domain analysis to be organised in a way that is easy to follow and suitable for effective retrieval of information.

The taxonomy of the process plant sub-domain is shown in Figure 7-11. The moment the domain taxonomy is built, reusable components are easily allocated and their applications are identified simply by identifying the domain scope.

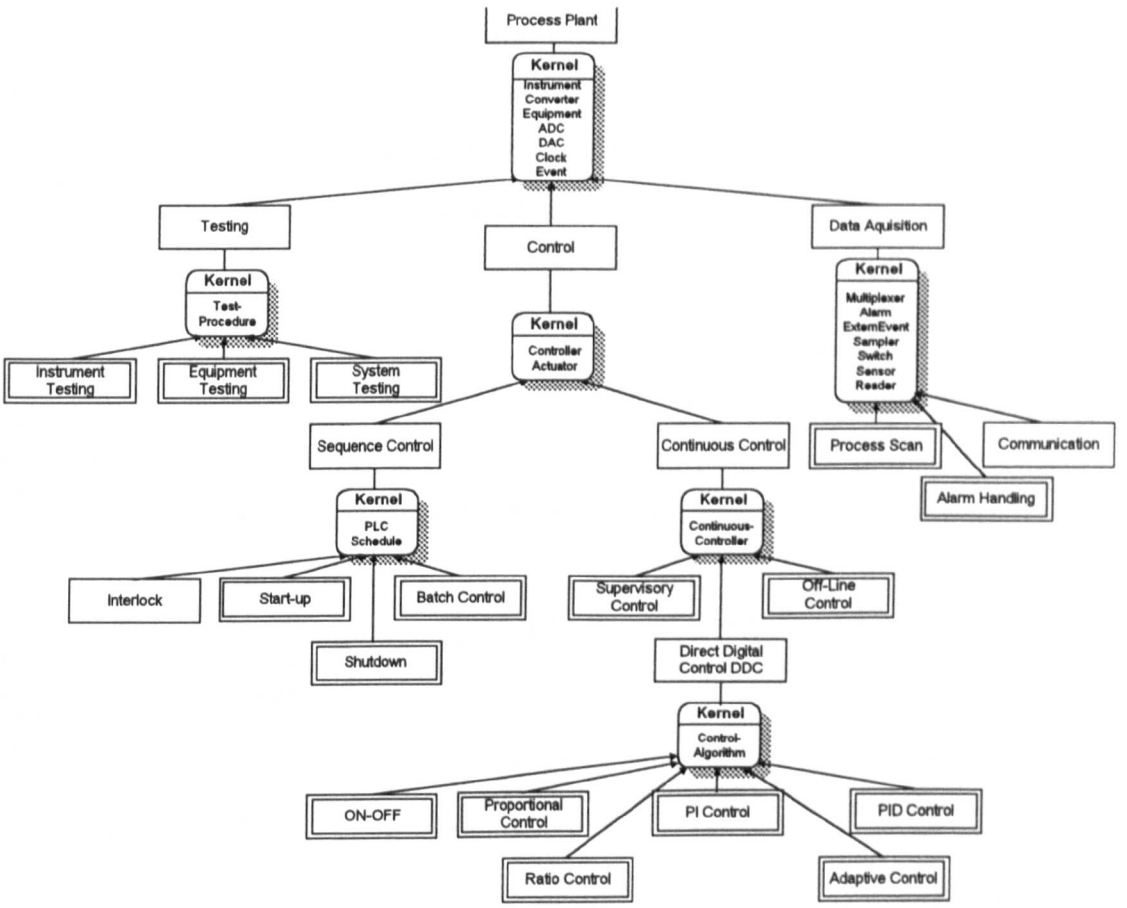

Figure 7-11 Process Plant Domain Taxonomy

The domain reference architecture contains information about how the components are related and design alternatives in terms of architecture schemas. In this example, the reference architecture within the control sub-domain is illustrated. The static relationships in the reference architecture are shown in Figure 7-12. These relationships (in terms of generalisation-specialisation, association and aggregation models) represent dependencies among components. Some design alternatives are shown in Figure 7-12. For instance, the *Control_Algorithm* component is a polymorphic non-autonomous component which is linked by a *whole-part*

relationship with a semi-autonomous component *(Continuous-Controller)*. The Control-Algorithm component in this case is an abstract object that represents a design alternative. Such an architecture schema may be instantiated into a specific system design using one of the *Control_Algorithm* children components. If a specific design conception requires an explicit modelling then concrete objects should be used in the schema as the case with the three components *P_Control, PI_Control* and *PID_Control,* which are linked by an EX-ORed whole-part relationship with *Continuous-Controller* component.

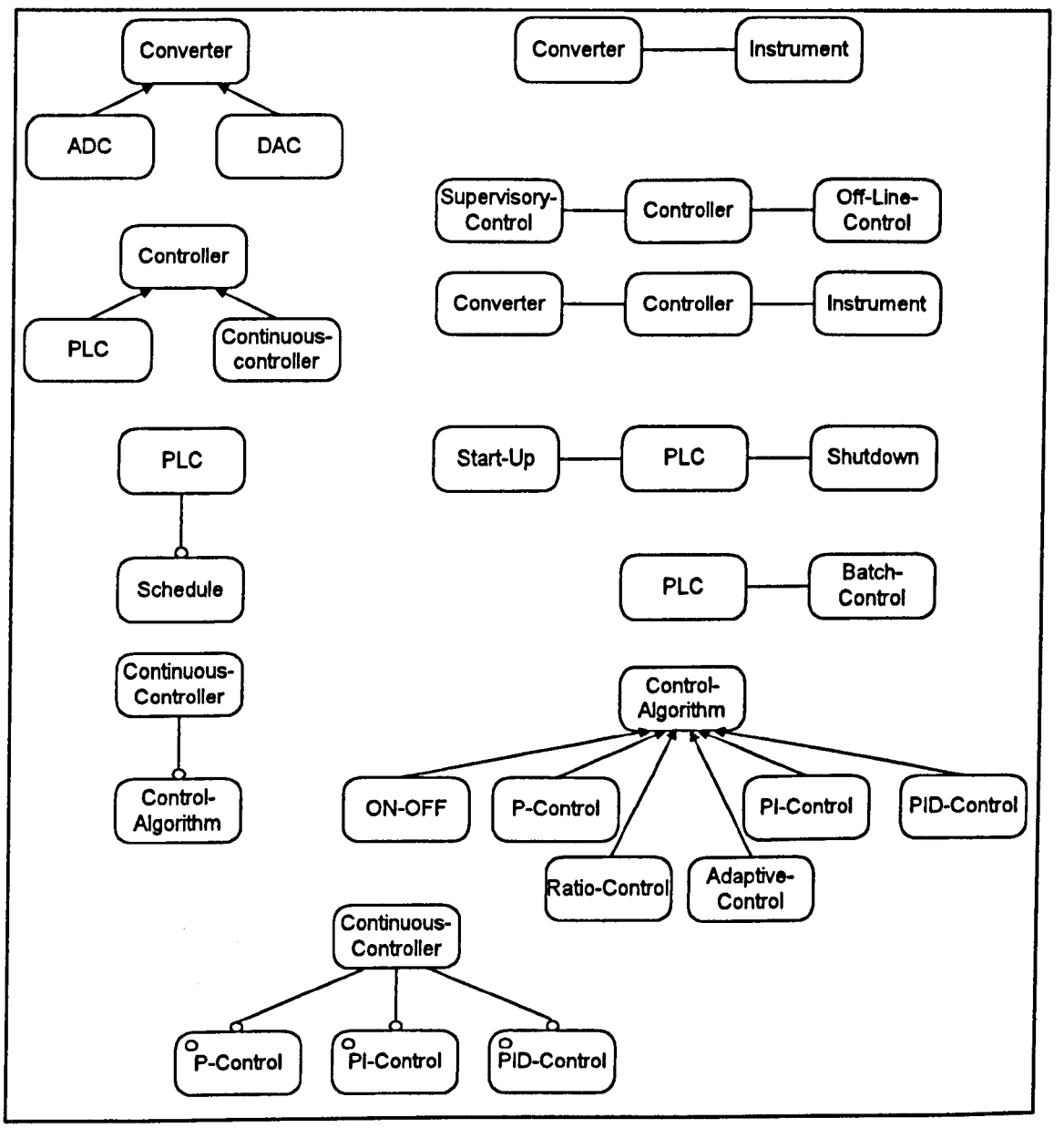

Figure 7-12 Static Relationships within Control Sub-domain

In dynamic relationships, interactions between components are modelled which are used to represent conceptual design decisions of a certain domain transaction. Such domain transactions are captured in one or more domain scenarios and restricted by domain rationales. In the control sub-domain scope, a typical domain scenario is illustrated in Figure 7-13. This scenario represents the transaction of reading a process value by the *Controller* component for computing a controller output value. Such a scenario could be modelled using the dynamic relationships in the generic architectures. A similar scenario in the control sub-domain scope is shown in Figure 7-13; the controller output scenario. This scenario is a typical domain specific scenario within the process control domain which specifies how the controller performs its control actions and interacts with the process.

Reading Process Value Scenario

In order to compute the controller output, the controller reads a process value from the relevant instrument. This is an analogue quantity that needs to be converted into a digital quantity by means of a suitable analogue-to-digital converter. The period of the reading frequency is determined by a real-time clock.

Controller Output Scenario

The controller carries out a control action computation using a specific control algorithm. The controller output value depends upon the controlled value, the controller last value and the control-algorithm parameters. When the controller action is computed, the controller output value is sent to an actuator for modifying the process state. Typically, the controller output value is first converted into an analogue value (using a digital-to-analogue converter) before it is sent to the actuator. The actuator then performs the controller action on the process.

Figure 7-13 Domain Scenarios

Figure 7-14 shows a number of dynamic relationships to model the scenarios shown in Figure 7-13 and other domain specific transactions. The architecture schema shown in Figure 7-14-A models the first scenario, whereas Figure 7-14-B models the second scenario. In Figure 7-14-C a batch process model is used in an architecture schema for modelling the shutdown procedure in the domain. The shutdown procedure usually comprises a certain schedule for switching off a number of equipment (motors, pumps, compressors ...etc.) in a certain sequence. The shutdown component is triggered by an external event, and in turn it passes a message to the *PLC* component for switching off the equipment. This process is a periodic process which means that a number of equipment and a number of steps in the PLC program are involved. The batch process ceases when the end of the schedule is reached. An event-driven model is used in the schema to mark the end of the schedule as an event for stopping the shutdown process.

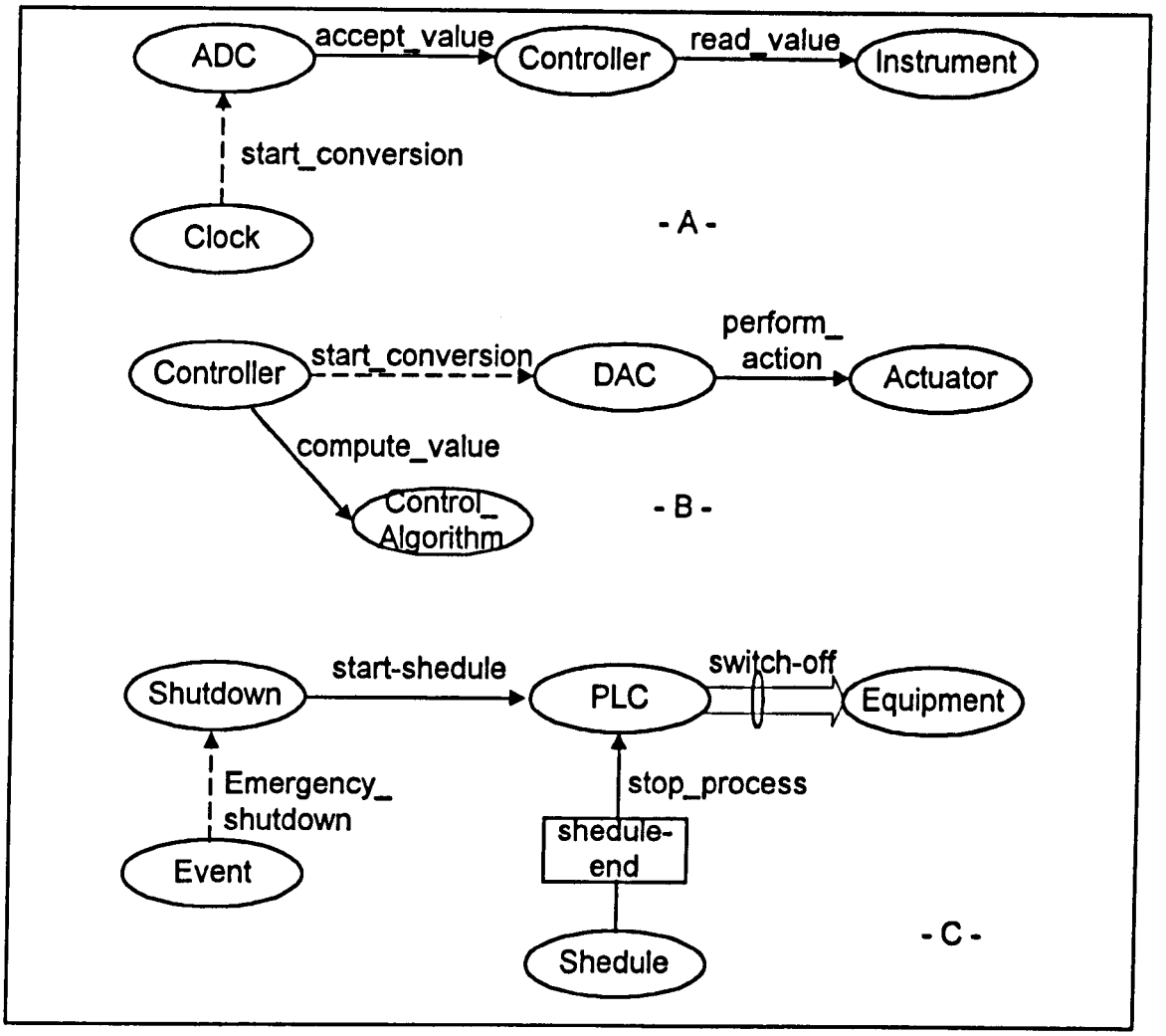

Figure 7-14 Reference Architecture Dynamic Relationships

The modelling process continues for modelling all the relationships between the components in the domain model. In this case study, only the domain artefacts are modelled. Domain resources are also added to the domain model as well such as the scenarios shown in Figure 7-13. Specifications of some of the components used in the above architecture schemas are found in Appendix B.

7.6 Summary

In this chapter, a number of automated tools for modelling and retrieving domain assets were described. As outlined in the previous sections, the tools were based on the existence of a common knowledge base of the domain resources and artefacts. This knowledge base is central to the DOOR approach as well as the tools. An enumerated classification scheme

has been adopted for storing and retrieving the domain assets. Therefore, when components (as well as other information) within the domain are stored and retrieved the scope of their application is specified as abstraction levels within the domain taxonomy. On the other hand, interactions between components and architecture schemas are governed by the scope of the participating components. This is checked with regards to the constraints specified by the generic architectural models (see chapter five), which are performed automatically when architecture schemas are processed internally.

A case study from the domain of process control was introduced to illustrate how this approach could be used for modelling a real world domain. One of the main conclusions drawn from analysing such a domain is the amount of information involved in the analysis process. Such information needs to be sorted and organised in such a way to allow easy locating and trace as well as retrieval of the domain assets. Using the DOOR approach gives us a more systematic way of organising and representing the domain knowledge that are easy to follow and understand. For instance, the preliminary identification of reusable components in the domain (Appendix B) results in a list of components that has no guarantee for reuse. The moment that these components are classified according to their scopes, they have better chance to be found and retrieved since their application is narrowed. With the specification of the relationships between components, the components are given an extra dimension in terms of behaviour that provides better grounds for understandability as well as tracing and retrieving them.

As a summary, DOOR approach provides a solution for some of the problems associated with reuse that have been outlined in the first chapter. There are some limitations to using this approach which will be discussed in the next chapter.

PAGE
MISSING
IN
ORIGINAL

Chapter Eight

8. Conclusions, Critical Assessment and Future Work

8.1 Introduction

In this chapter, the DOOR approach is evaluated to identify its advantages and limitations and suggestions for future work are highlighted. Section 8.2 starts with analysing the results of applying the approach pointing out its strengths by comparing it to other approaches. The limitations are then outlined and situations where the approach could or could not be applied are identified. In section 8.3, possible areas for improving and expanding the approach are set out.

8.2 Conclusions and Critical Assessment

The thesis has described an approach to software development from reusable objects, which is called domain-oriented object reuse (DOOR). The approach has tackled the problems associated with reusing objects within the context of a specific domain. The main features of the approach may be summarised in the following points:

1. DOOR integrates Domain analysis within the development life-cycle and allows the domain assets to evolve as new systems are analysed and implemented.
2. The context for reuse is emphasised in DOOR by identifying the scope of the analysis of domain knowledge as well as the application of the reusable assets.
3. An enumeration classification scheme is adopted for organising the domain knowledge as well as locating and retrieving its assets. The scheme is based on the notion of the domain scope. Thus, scope acts both as a style for

presenting the domain knowledge and a technique for archiving its components.

4. Reusable components are described using the 3-D model which specifies components in terms of their scope, behaviour and reaction space. The model is used to describe the components' functionality. Together with components' type, the model is used to specify the interactions between components within the domain scope.
5. Dependencies between components are modelled by means of a number of architectural models which are defined in the generic software architectures. Each model comprises a relationship which is used to link two or more components together and a set of constraints which restrict the use of the model to specific component types.
6. Component relationships are specified using architecture schemas which represent design conceptions or design alternatives within the domain. When systems are built, the architecture schemas are instantiated into specific system design decisions. Thus the architectural models and schemas present solutions to the domain problems, which (the solutions) are encapsulated in the reusable components.
7. In addition to the technical support for reuse provided by the generic software architectures, DOOR provides a reuse process for modelling and applying the domain assets. The process has been divided into two phases, Domain Engineering and Application Engineering. The process is organised in a way that supports the evolution of the domain model and its artefacts. This is achieved by taking into consideration the domain-specific features and requirements that may be identified when new systems are analysed and developed.
8. A set of guidelines has been introduced for guiding the domain engineer in building the domain artefacts and validating the domain model design. A number of guidelines have been proposed for assessing the process and identifying possible means for updating the domain model in terms of new artefacts or modifying the existing ones.

9. The approach is supported by an integrated set of tools to help in the organisation of domains and modelling their assets. The tools were based on the existence of a common knowledge base of the domain resources and artefacts.

8.2.1 Advantages of the DOOR approach

In our opinion, the strengths of the approach could be summarised in the following points:

1. The enumerated classification scheme adopted in this approach is used for presenting the domain knowledge in a structural form based on domain abstractions. This type of classification organises the domain in a hierarchy of related sub-domains which makes it easier for the user to locate and retrieve information effectively.
2. When a system is built within a certain sub-domain scope, reusable components are located immediately and presented to the user. A number of levels of component reusability could be identified depending on their scope. Components within a specific domain scope have a high chance for reuse within the scope of its subordinate; thus such components have the highest level of reusability.
3. When components are specified and included in the domain model the relationships between components are also modelled. These relationships represent some design conceptions or alternatives in terms of architecture schemas that could be instantiated into design decisions when systems are built. The same components may be used in more than one schema allowing reconfiguration of the architecture for accommodating different design conceptions. This increases the reusability of the component in different contents.
4. Using the generic software architectures in modelling component relationships has the advantage of standardising the types of relationships that link components together. A software practitioner needs only to be familiar with a limited number of architectural models for modelling or understanding components relationships.

5. The generic architectures impose some constraints on component relationships within the architectural models. Although such constraints will restrict the use of the models to certain types of components, it has the advantage of allowing checking of the architecture design against the constraints. Another advantage is that the constraints are used for ensuring that the relationships used in a certain schema are the ones that are intended for domain-specific constraints.
6. The use of the guidelines in the DOOR reuse process is very useful for anyone who is using the approach as they provide a number of steps to be followed for achieving the complete domain model. They are also useful for checking of the architecture schemas automatically.

8.2.2 Limitation of DOOR approach

Despite the advantage that has been listed in the previous section DOOR has certain limitations. Mainly these limitations are associated with the hierarchical organisation of the domain assets. The following points summarise these limitations:

1. The hierarchical approach to organising the domains in a taxonomy tree might not suit all domains. Some domains may not be so structured, some are just flat domains or some of them are so inter-related that their sub-domains are dependent on each others' operations. In such cases DOOR might not be the best way of organising the domain taxonomy.
2. In some cases domains cross each other's borders in which components or operations in one domain could be used in a different domain. Currently, DOOR does not support importing components from external domains to be used by the domain components. This is not a limitation but a case for possible extension for the approach.
3. Sometimes problems related to interaction between sub-domains in the taxonomy are solved by introducing common components within the parent domain scope. Such components are then linked to components within the scope of the sub-domains, mainly by an **is-a** relationship (as shown in Figure 8-1) which allows the components in the sub-domain to access the parent

component. In some cases the interaction between components happens within sub-domains that are not located in the subordinate of one parent domain in which case the is-a relationships becomes more and more complicated which hinders the reuse of the components in the domain model.

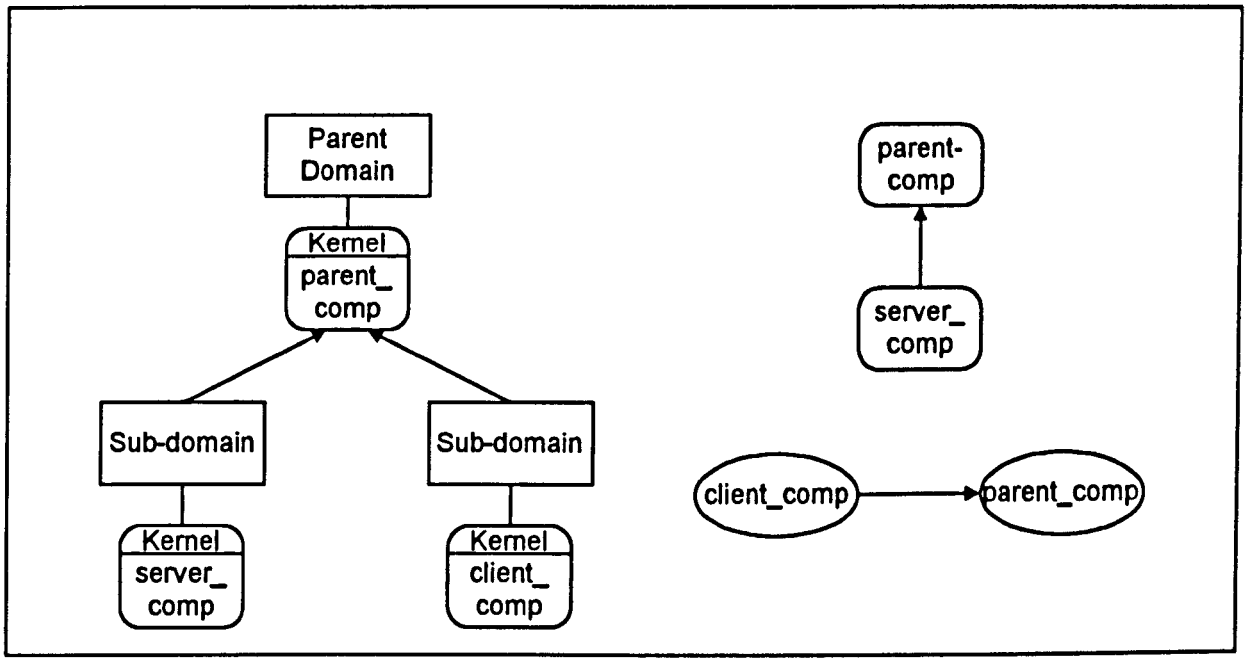

Figure 8-1 Cross Level Component Interaction

4. DOOR allows components in the domain model to evolve with time as new systems are built within the domain scope. In DOOR the components in the domain model are assumed to be the final versions in the domain model. In some cases we need to keep a track of all the changes made to a specific component in order to understand the component behaviour or for reusing older versions of the component. Currently DOOR does not support multiple versions of reusable components nor does it support multiple facets of components. The only facets supported by DOOR are the components' 3-D model, specifications and code.

8.3 Some Ideas for Future Work

During the course of this project, a number of problems have been addressed regarding reuse of software components and analysing and presenting domain knowledge. Further problems have been faced as the work progressed where we tried to give some feasible solutions through the use of generic software

architectures. Some of these problems still need to be addressed where time has not been enough to do so during this project. Other ideas have been inspired by the research which could be pursued as future extension to the project or as a whole new projects in the future. The following ideas are possible future work within the area of object-oriented reuse and domain specific architectures:

1. The first extension in this approach is to address the problem of inter-related domains where components specified in one domain may be used in another domain subject to some type and constraint checking. One possible solution to solve this problem is to allow the definition of an external domain or a friend domain (similar to a friend class in C++). In this case some research is needed to identify situations where such friend domains may or may not be permitted. Guidelines are also needed for specifying how external (or friend) domains are specified, in which case the guidelines proposed in this project need to be extended to accommodate this case.
2. The use of generic software architectures has prompted situations where a number of models could be used in one architectural schema. In situations like this, research could be conducted for identifying patterns of relationships within the architectural schemas. When such patterns are identified and classified, another possible future research is finding ways to convert these patterns into object-oriented patterns (see chapter three for discussion about object-oriented design patterns). Figure 8-2 shows a proposed approach for future project within the area of object-oriented reuse and pattern languages.

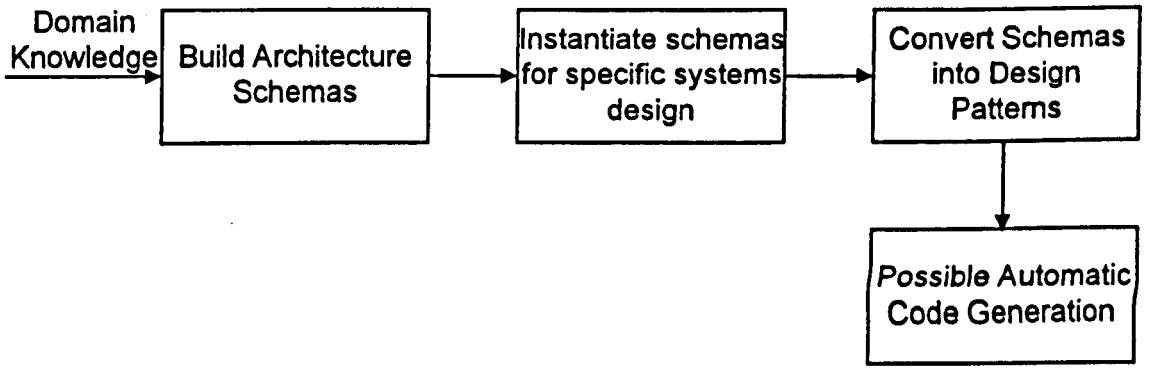

Figure 8-2 Generic Architecture to Design Patterns

As shown in Figure 8-2, the proposal comprises the use of the generic architectures to build architectural schemas that encapsulate the domain and abstraction and could be instantiated into specific system's architecture or design based on the architectural models. Such system architecture is then mapped into a number of object-oriented design patterns which could provide more implementation oriented solution to the domain knowledge. Possible automatic generation of code could then be investigated based on specific design patterns. A pattern language might be needed for specifying the system design in terms of design patterns; such a language could be used as a basis for code generation.

3. One of the problems that are related to reusing and retrieving components is identifying which version of the component is required and what modification has been made on the component so far. A highly needed research work at the moment is investigating configuration management procedures related to indexing and maintaining reusable components. Such configuration management procedures could be more useful if supported by a software tool that automates version control of the reusable components. Such a tool must have the ability to express the type of changes, the motivation behind the change and the new features as well as the old features. Links from new to old versions and vice versa should be allowed within the tool in order to facilitate trace the relevant component in the domain model.

4. One of the increasingly used approaches to software reuse is the use of the internet and the World Wide Web (WWW) for tracing and retrieving software components. In our approach a Hyper Text Mark-up Language (HTML) files have been automatically generated for supporting accessing and retrieving domain resources from the domain model. A possible extension of this work could involve the use of the internet as an environment for accessing libraries of components with procedures for subscription to the library and access control for allowing multiple levels of access to the library according to domain abstractions. Internet support could be provided to

access the domain architectures and component relationships graphically. Links between components should also be traceable through the internet while having the access control still enforced on the library access by external subscribers.

As a conclusion, the work has addressed a number of the issues that are currently facing the reuse community and opened avenues for further work. One of the lessons which have been learnt in this project is: software components could be large or small, however their reusability depends on how well they are described and modelled. This leads to the issue of information presentation; the project has emphasised this point through presenting domain knowledge and domain-specific design conceptions. On the other hand, dedicated components that are designed to be applied in conjunction with other components could prove to be highly reusable because of their interaction with other components.

Finally, the approach has been applied to two case studies which showed benefits in modelling and retrieving domain assets.

Bibliography

Aggresti, W.W, 1986, "What are the New Paradigms?", *New Paradigms for Software Development,* ed. W.W. Agresti, IEEE press, pp. (6-10).

Agrwala, A., Krause, J. and Vestal, S., 1992, "Domain-Specific Architectures for Intelligent Guidance, Navigation and Control",in *Mettala, E. and Graham, M.H. (eds.), "The Domain Specific Software Architecture Program"*, Special Report, Software Engineering Institute, Carnegie Mellon University, Pittsburg, Pennsylvania 15213, CMU/SEI-92-SR-9, June 1992.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I. and Angel, S, 1977, "A Pattern Language", *Oxford University Press,* New York.

Al-Yasiri, A and Ramachandran, M. 1994, "Developing Software Systems with Domain Oriented Reuse", *EuroMicro 94,* Liverpool, UK, 5-8 September 1994

Anderson, B. 1994, "OOPSLA'93 Workshop Report, Patterns: Building Bloacks for Object-Oriented Architectures", *ACM SIGSOFT Software Engineering Notes,* Vol. 19, No. 1, January 1994, pp. 47-49.

Baker, T.P. and Scallon, G.M., 1986, "An Architecture for Real-Time Software Systems", *IEEE Software,* May 1995, pp. 51-58.

Biggerstaff, T.J. and Richter, C. 1987, "Reusability Framework, Assessment and directions", *IEEE Software,* Vol. 4, No. 2, March 1987.

Boehm, B., 1988, "A Spiral Model of Software Development and Enhancement", *IEEE Computer,* Vol. 21, No 5, May 1988, pp. (61-72).

Booch, G. 1986, "Software Engineering With Ada, 2nd Edition", *The Benjamin/ Cummings Publishing Company, Inc.*

Booch, G. 1987, "Software Components with Ada - Structures, Tools and Subsystems", *The Benjamin/ Cummings Publishing Company, Inc.*

Booch, G. 1993, "Patterns", *Object Magazine,* July-August, 1993, pp. 24-27.

Burton, B.A., Aragon, R.W., Baily, S.A., Koehler, K.D., and Mayes, L.A., 1987, "The Reusable Software Library", *IEEE Software,* July 1987, pp. (25-33).

Basili, V.R., Caldiera, G., McGarry, F., Pajerski, R., Page, G. and Waligora, S., 1992, "The software engineering laboratory - an operational software experience" *proceedings 14th international Conference Software Engineering,* Melbourne, Austrailia, May 1992, pp. (370-381).

Basili, V.R. and Green, S., 1994 , "Software Process Evolution in the SEL", *IEEE Software,* July 1994, pp. (58-66).

Biggerstaff, T.J. and Perlis, A.J. 1989, "Software Reusability -Volume I, Concepts and Models", *Addison-Wesley publishing company.*

Biggerstaff, T.J. and Perlis, A.J. 1989, "Software Reusability -Volume II, Applications and Experience", *Addison-Wesley publishing company.*

Braun, C., Hatch, W., Ruegsegger, T., Balzer, B., Feather, M, Goldman, N. and Wile, D., 1992, "Domain-Specific Architectures: Command and Control", in *Mettala, E. and Graham, M.H. (eds.), "The Domain Specific Software Architecture Program"*, Special Report, Software Engineering Institute, Carnegie Mellon University, Pittsburg, Pennsylvania 15213, CMU/SEI-92-SR-9, June 1992.

Caldiera, G. and Basili, V.R., 1991, "Identifying and Qualifying Reusable Software Components", *IEEE Computer,* February 1991, pp. (61-70).

Carter, R.J. 1990, "The Form of Reusable Ada Components for Concurrent Use", *Ada Letters,* Vol. X, No. 1, January/ February 1990 pp. (118-21).

Chen, P.S.S., Henniker, R., and Jarke, M., 1993, "On the Retrieval of Reusable Software Components", Procedings of 2nd *International Workshop on Software Reusability,* , 1993 (REUSE'93), March 24--26 1993, Lucca, Italy, pp. (99-108).

Coad, P. 1992, "Object-Oriented Patterns", *Communications of the ACM,* Vol. 35, No. 9, September 1992, pp 152-9.

Coad, P. and Yourdon, E., 1991, "Object-Oriented Analysis", Second Edition, *Prentice Hall, Englewood Cliffs, N.J.*

Coglianese, L., Goodwin, M., Smith, R., Tracz, W., Bator, D., Bellman, K., Gries, D., McAllester, D., Selb, R. and Taylor, R, 1992, "An Avionics Domain-Specific Software Architecture", *in Mettala, E. and Graham, M.H. (eds.), "The Domain Specific Software Architecture Program"*, Special Report, Software Engineering Institute, Carnegie Mellon University, Pittsburg, Pennsylvania 15213, CMU/SEI-92-SR-9, June 1992.

Dutton,G. and Sims, D., 1994, "Patterns in OO Design and Code Could Improve Reuse", *IEEE Software,* May 1994, pp 101-3.

Fafchamps, D. 1994, "Organisational Factors and Reuse", *IEEE Software,* September 1994, pp. (31- 41).

Frakes, W. and Gandel, P., 1990, "Presenting Reusable Software", *Information and Software Technology.*

Frakes, W.B. and Najmeh, B.A., 1987, "An Information System for Software Reuse", *in Software Reuse : Emerging Technology,* Editor: W. Tracz, IEEE CS Press pp. 142-151.

Frakes, W.B. and Pole, T., 1992, "An Emperical Study of Representation Methods for Reusable Software Components", *Technical Report, Software Productivity Consortium,* Herndon, Va. May 1992.

Gamma, E.; Helm, R.; Johnson, R. and Vlissides, J. 1993, "Design Patterns: Abstraction and Reuse of Object-Oriented Design", Published as lecture notes in *Computer Science,* Vol.707, pp 406-431.

Gamma, E.; Helm, R.; Johnson, R. and Vlissides, J. 1995, "Design Patterns: Elements of Reusable Object-Oriented Software", *Addison Wesley Publishing Company.*

Garlan, D. (ed.) 1995, *Proceedings of the First International Conference on Software Architecture,* (Apri 1995). CMU Tach Rep. CMU-CS-95-131.

Garlan, D., Allen, R. and Ockerbloom, J., 1995, "Architectural Mismatch: Why Reuse Is So Hard", *IEEE Software,* Nov. 1995, pp. 17-26.

Garlan, D. and Perry, D. (eds) 1995, "Special Issue on Sotware Architecture", *IEEE Transactions on Software Engineering,* Nov. 1995.

Garlan, M.Shaw, C. Okasaki, C. Scott, and R. Swonger, 1992 "Experience with a Course on architectures for Software Systems" *Proceedings of the sixth SEI Conference on Software Engineering Education*, Springer Verlag, LNCS 376, October 1992.

Garlan, D. and Shaw, M., 1993, "An Introduction to Software Architecture", in *Advances In Software Engineering and Knowledge Engineering*, Vol. 2, edited by: Abriola, B. and Tortor, G., World Scientific Publications, Singapore, 1993, ISBN 981-02-1594-0.

Gautier, R.J. and Wallis, P.J.L 1990 "Software Reuse with Ada" *Peter Peregrinus Ltd.*, London, United Kingdom.

Gomaa, H., Fairl, R and Kerschberg, L, 1989, "Towards an Evolutionary Domain Life Cycle Model", *Proceedings of the Workshop on Domain Modelling for Software Engineering*, OOPSLA'89, New Orleans, October 1989.

Gomaa, H., 1992, "An Object-Oriented Domain Analysis And Modelling Method For Software Reuse", *Proc. Hawaii International Conference on System Sciences, Hawaii,* January 1992, pp. 46-56.

Gomaa, H., 1993a, "A Reuse-Oriented Approach For Structuring and Configuring Distributed Applications", *Software Engineering Journal,* March 1993, pp.61-71.

Gomaa, H., 1993b, "Software Design Methods for Concurrent and Real-Time Systems", *Addison-Wesley Publishing Company.*

Graham, I., 1991, "Object Oriented Methods", *Addison-Wesley Publishing Company*, 1991.

Guttag, J.V., Horning, J.J., and Wing, J.M., 1985, "An Overview of the Larch Family of Specification Languages", *IEEE Software,* Vol. 2, No. 5, Sept. 1985, pp. 24-36.

Halang, W.A. and Kramer, B.J. 1994, "Safety Assurance in Process Control", *IEEE Software,* January 1994, pp. 61- 67.

Hall, P. and Boldyreff, C., 1991, "Software Reuse", *in Software Engineer's Reference Book,* Ed. J.A. McDermid, Butterworth-Heinemann Ltd., Oxford.

Hayes-Roth, B., 1985, "A Blackboard Architecture for Control", *Artificial Intelligence*, Vol. 26, pp. 251-321.

Hayes-Roth, B., Pfleger, K., Lalanda, P., Morignot, P. and Balabanovic, M., 1995, "A domain Specific Software Architecture for Adaptive Intelligent Systems", *IEEE Transactions on Software Engineering*, Vol. 21, No. 4, April 1995, pp. 288-301.

Hayes-Roth, F., Erman, L.D., Terry, A. and Hayes-Roth, B., 1992, "Domain-Specific Software Architectures: Distributed Intelligent Control and Communication", in *Mettala, E. and Graham, M.H. (eds.), "The Domain Specific Software Architecture Program"*, Special Report, Software Engineering Institute, Carnegie Mellon University, Pittsburg, Pennsylvania 15213, CMU/SEI-92-SR-9, June 1992.

Henderson-Sellers, B. amd Edwards, J.M., 1993, "The Fountain Model for Object-Oriented System Development", *Object Magazine*, July-August 1993, pp. (71-79).

Horowitz, E. and Munson, J.B. 1984, "An Expensive View of Reusable Software" *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September 1984.

Hutchinson, J.W. and Hindley, P.G., 1988, "A Preliminary Study of Large-Scale Software Reuse", *Software Engineering Journal*, September 1988, pp. (208-12).

Johnson, R.E. and Foote, B. 1988, "Designing Reusable Classes", *Journal of Object-Oriented Programming*, Vol. 1, No. 2, June/July 1988, pp (22-35).

Johnson, R.E. 1994, "Why a Conference on Pattern Languages?", *ACM SIGSOFT Software Engineering Notes*, Vol. 19, No. 1, January 1994, pp 50-52.

Kazman, R. and Bass, L. 1994, "Towards Deriving Software Architectures From Quality Attributes", Technical Report, *Software Engineering Institute, Carnegie Mellon University*, Pittsburg, Pennsylvania 15213, CMU/SEI-94-TR-10, ESC-TR-94-010, June 1992.

Kazman, R., Bass, L., Abowd, G. and Webb, M., 1994. "SAAM: A Method for Analyzing the Properties of Software Architectures", *Poceedings of the*

Internatinonal Conference on Software Engineering, Sorrento, Italy, May 1994, pp. 81-90.

Kogut, P. and Clements, P. 1994, "The Software Architecture Renaissance", *CrossTalk,* November 1994, pp 20-3.

Kruchten, P.B., 1995, "The 4+1 View Model of Architecture", *IEEE Software,* Nov. 1995, pp. 51-60.

Lanergan, R.G. and Grasso, A., 1984, "Software Engineering with Reusable Designs and Code", *IEEE Transactions on Software Engineering,* Vol. SE-10, No. 5, September 1984, pp (498-501).

Leveson, N.G. 1990, "The challenge of Building Process-Control Software", *IEEE Software,* November 1990, pp. 55-62.

Leveson, N.G., Heimdahl, M.P.E., Hildreth, H. and Reese, J.D., "Requirements Specification for Process-Control Systems", *IEEE Trans. on Software Engineering,* Vol. 20, No. 9, September 1994, pp. 684-707.

Lim, W.C., 1994, "Effects of Reuse on Quality, Productivity and Economics", *IEEE Software,* September 1994, pp (23-30).

Lubars, M.D., 1991, "Domain Analysis and Domain Engineering in IDeA", *Domain Analysis and Software Systems Modelling,* IEEE Computer Society Press, 1991, pp. 163-178.

Maarek, Y. S, 1993, "Introduction to Information Retrieval for Software Reuse", in *Advances In Software Engineering and Knowledge Engineering,* Vol. 2, edited by: Abriola, B. and Tortor, G., World Scientific Publications, Singapore, 1993, ISBN 981-02-1594-0.

Matsumoto, Y., Sasaki, O., Nakajima, S., Takezawa, K., Yamamoto, S. and Tanaka, T., 1980, "SWB System: A Software Factory", *In Software Engineering Environments* (Editor Huenke) North-Holland, pp. (305-18).

Matsumoto, Y. 1984, "Some Experiences in Promoting Reusable Software Presentation in Higher Abstraction Levels", *IEEE Transactions on Software Engineerig,* Vol. SE-10, No. 5, September 1984.

Matsumoto, Y. 1993, "Experiences from Software Reuse in Industrial Process Control Applications", *Proc. of the Second International Workshop on Software Reusability,* 1993 (REUSE'93), March 24-26 1993, Lucca, Italy, pp.186-195.

McGregor, J.D. and Sykes, D.A. 1992, "Object-Oriented Software Development: Engineering Software for Reuse", *Van Nostrand Reinhold,* New York, USA.

Mettala, E. and Graham, M.H. 1992, "The Domain Specific Software Architecture Program", Special Report, *Software Engineering Institute, Carnegie Mellon University,* Pittsburg, Pennsylvania 15213, CMU/SEI-92-SR-9, June 1992.

Mili, A., Mili, R., and Mittermeir, R., 1994, "Storing and Retrieving Software Components: a refinement-based approach", *Proceedings of 16th International Conference on Software Engineering,* Sorrento, Italy, May 1994.

Mili, H., Mili, F. and Mili A., 1995, "Reusing Software: Issues and Research Directions", *IEEE Transactions on Software Engineerng,* Vol. 21, No. 6, June 1995, pp. (528-62).

Moore, J.M. and Bailin, S.C., 1991, "Domain Analysis: Framework For Reuse", *Domain Analysis and Software Systems Modelling,* IEEE Computer Society Press, 1991, pp. 179-203.

Morel, J-M., Faget, J. 1993, " The REBOOT Environment", *Proc. of the Second International Workshop on Software Reusability,* 1993 (REUSE'93), March 24-26 1993, Lucca, Italy, pp. (80-88).

Neighbors, J.M., 1980, "Software Construction using Components" Ph.D. Thesis and Technical Rport TR-160, *University of California, Irvine, ICS Dept.*

Neighbors, J.M., 1984, "The Draco Approach To Constructing Software From Reusable Components", *IEEE Trans. on Software Engineering,* Vol. SE-10, No. 5, Sept. 1984, pp. 564-574.

Neighbors, J.M. 1989, "DRACO: A Method for Engineering Reusable Software Systems", in Softwar Reusability Vol. I (eds: Biggerstaff and Perlis) *Association for Computing Machinery Presss.*

Peters, L. 1987, "Advanced Structured Analysis and Desig", ed. R. W. Jensen, Prentice Hall.

Pirklbauer, R.P. and Weinreich, R., 1994, "Object-Oriented Process Control Software", *Journal of Object-Oriented Programming,* May 1994, pp. 30-35.

Prieto-Díaz, R. 1987, "Classifying Software For Reuse", *IEEE Software,* Vol. 4, No, 1, January, 1987.

Pressman, R.S., 1992, "Software Engineering A Practitioner's Approach, Third Edition", *McGraw-Hill International Inc.*

Prieto-Díaz, R. and Freeman, P., 1987, "Classifying Software For Reusability", *IEEE Software,* Vol. 4, No, 1, January, 1987, pp. (6-16).

Prieto-Díaz, R., 1990, "Domain Analysis: An Introduction", *ACM SIGSOFT Software Engineering Notes,* Vol. 15, No. 2, Apr. 1990, pp. 47-54.

Ramachandran, M. and Al-Yasiri, A. 1994A, "Reusing and Retrieving Software Components: An Object-Oriented Domain Analysis Approach", *OOIS' 94 International Conf. on Object-Oriented Information Systems,* London, UK, 19-21 December 1994.

Ramachandran, M. and Al-Yasiri, A. 1994B, "An Object-Oriented Domain Analysis Method", *Object Technology 94,* Oxford, UK, 28-30 March 1994.

Sikkel, K. and van Vliet, J.C., 1992, "Abstract Data Types as Reusable Software Components: the Case for Twin ADTs", *Software Engineering Journal,* May 1992, pp. (177-183).

Shaw, M., 1990, "Toward Higher-Level Abstractions for Software Systems", *Data and Knowledge Engineering,* Vol. 5, North Holland: Elsevier Science Publishers B.V., pp. 119-128.

Shaw, M. 1994, "Comparing Architectural Design Styles", *IEEE Software,* Nov.1995, pp. 27-41.

Shaw, M., 1995, "Beyond Objects: A Software Design Paradigm Based on Process Control", *ACM SIGSFT Software Engineering Notes,* Vol. 20, No.1, Jan. 1995, pp. 27-38.

Shlaer, S. and Mellor, S., 1989, "An Object-Oriented Approach to Domain Analysis", *ACM SIGSOFT Software Engineering Notes,* Vol. 14, No. 5, July 1989 pp. (66-77).

Shlaer, S. and Mellor, S., 1993, "A Deeper Look at the Transition from Analysis to Design", *Journal of Object Oriented Programming*, February 1993, pp.(16-21).

Smith, J.D., 1990, "Reusability & Software Constructions C & C++", *John Wiley and Sons.*

Sommerville, I. 1992, "Software Engineering", Fourth Edition, *Addison-Wesley Publishing Company, Inc.*

Sørumgård, L.S., Sindre, G. and Stokke, F. 1993, "Experiences from Application of a faceted Classification Scheme", *Proc. of the Second International Workshop on Software Reusability,* 1993 (REUSE'93), March 24--26 1993, Lucca, Italy, pp. (502-12).

Tajima, D. and Matsubara, T. 1984, "Inside the Japanese Software Industry", *IEEE Computer*, March 1984, pp. (34-43).

Terry, A., Hayes-Roth, F., Coleman, N., Devito, M., Papanagapoulos, G. and Hayes-Roth, B., 1994, "Overview of Teknowledge's Domain-Specific Software Architecture Program", *ACM SIGSOFT Software Engineering Notes*, Vol. 19, No. 4, October 1994, pp. 68-76.

Tracz, W. (Editor), 1994, "Domain-Specific Software Architecture (DSSA) Frequently Asked Questions (FAQ)", *ACM SIGSOFT Software Engineering Notes"*, Vol. 19, No. 2, April 1994, pp. 52-6.

Tracz, W. 1995, "DSSA (Domain Specific Software Architecture) Pedalogical Example", *ACM SIGSOFT Software Engineering Notes*, Vol. 20, No. 3 July 1995, pp. 49-62.

Van Der Linden, F.J. an Müller, J.K. 1995, "Creating Architectures with Building Blocks", *IEEE Software,* Nov. 1995, pp. 51-60.

Wartik, S. and Prieto-Díaz, R.,1992, "Criteria for Comparing Reuse-Oriented Domain Analysis Approaches", *International Journal of Software Engineering and Knowledge Engineering*, Vol. 2, No. 3, pp. (403-31).

Yoelle, Y.S., Maarek, S., Berry, D.M., and Kaiser, G.E., 1991, "An Information Retrieval Approach for Automatically Constructing Software Libraries", *IEEE Trans. on Software Engineering,* Vol. 17, No. 8, August 1991, pp.(800-13).

Zaremski, A.M. and Wing, J.M., 1993, "Signature Mtching: A Key to Reuse Software Engineering", *1st ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Vol. 18.

Appendix A

Description of the Theatre Domain Example

A-1 Introduction

In this appendix, the example of the theatre domain is described as presented by Tracz in his paper (Tracz 1995). The description focuses on the modelling aspects of the domain artefacts as presented in the paper. A user needs statement of a reservation system in the theatre domain is first introduced. Later this statement will be used as a basis for generalising the solution over the whole domain. Next, a number of scenarios from the domain are presented. Some of the domain model components are then shown as modelled in the paper.

A-2 User Needs Statement

> I am in charge of the finances for a play that is being performed by our community theatrical group. This is a one time shot, but I think it would be nice to have a computer program to help the person taking phone and mail orders for tickets. Depending on how it works, I may want to use it for the rest of the performances by our theatrical group.
>
> The theatre we are using has reserved seats (i.e., row number, seat number). We are charging $10 for orchestra seats and $7 for seats in the balcony.
>
> We would like the program to tell us such things as: how many tickets are sold, how many are left, and how much money has been taken in. To help the ticket agent, we also would like a display of the seating arrangement that shows which seats are sold and which are available.

A-3 Scenarios

The following scenarios consist of a list of numbered, labelled scenario steps or events followed by a brief description.

A-3-1 Ticket Purchase Scenario

1. **Ask:** The customer asks the agent what seats are available.
2. **Look:** The agent enters the appropriate command into his/her terminal and relates the results to the customer (cost, section, row number and seat number).

3. **Decide:** The customer decides what seats are desired, if any, and tells the agent.
4. **Buy:** The customer pays the agent for tickets. The agent gives the tickets to the customer.
5. **Update:** The agent records the transaction.

A-3-2 Ticket Return Scenario

1. **Return:** The customer gives the agent tickets that are no longer needed.
2. **Refund:** The agent gives the customer money back.
3. **Update:** The agent records the transaction.

A-3-3 Ticket Exchange Scenario

1. **Ask:** The customer asks the agent what seats are available.
2. **Look:** The agent enters the appropriate command into his/her terminal and relates the results to the customer (cost, section, row number and seat number).
3. **Decide:** The customer decides what seats are desired, if any, and tells the agent.
4. **Exchange:** The customer gives the agent the old tickets, then the agent gives the customer the new tickets.
   Depending on the price of the new tickets, the agent either collects additional money from the customer or issues a refund.
5. **Update:** The agent records the transaction.

A-3-4 Ticket Sales Analysis Scenario

1. **Stop Sales:** The sales manager enters the command to stop the sale of tickets for a particular performance.
2. **Tally:** The ticket sales program generates a report listing total sales.

A-3-5 Theatre Configuration Scenario

1. **Performance Logistics:** The sales manager enters in the name, time, location and date of the performance.
2. **Seating Arrangement:** The sales manager decides if the performance is "Reserved Seating or "Open Seating".
3. **Theatre Logistics:** If this performance is reserved seating, then the sales manager enters the numbered kind of sections in the

theatre, what rows are in what sections and what seats are in what rows.

If this performance is open seating, then the sales manager enters the total number of tickets to be sold.

4. **Pricing:** The sales manager enters in the price of each ticket, determined by section and seating style.

A-4 Domain Dictionary

The Following domain dictionary consists of examples of commonly used words and phrases found in the scenarios and customer needs document.

Agent: The person who interacts with the application, answers customer questions and handles tickets and money.

Balcony: The farthest away and usually the least expensive seats in the theatre.

Configuration: Information describing the performance and seating style for which tickets are sold. "see Performance and Seating Style".

Open Seating: A seating style where there are no reserved seats (a ticket is good for any seat in the theatre).

Orchestra: The closest and generally the most expensive seats in a theatre.

Performance: The date, time, location and name of a theatrical production.

Seating Style: Either open seating or reserved seating.

Theatre: The place full of named sections, rows and seats where performances are held.

Ticket: A ticket is what the customer buys, sells and uses to get in the door of a performance.

A-5 Theatre Object Aggregation and Taxonomy Diagrams

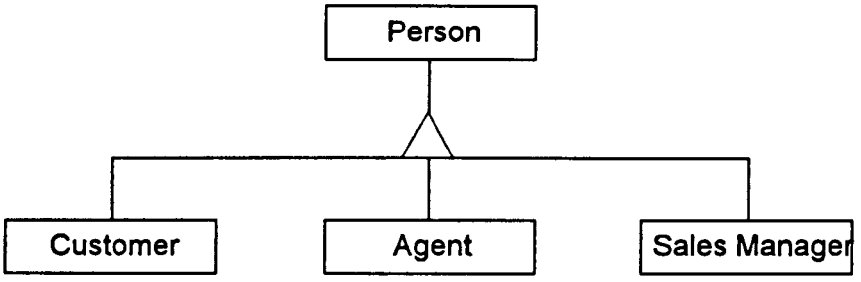

Figure-1 Person Types Taxonomy

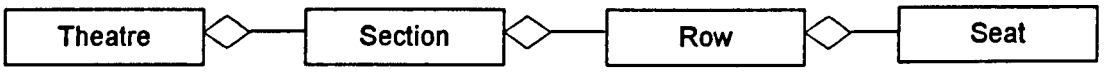

Figure-2 Theatre Aggregation Hierarchy

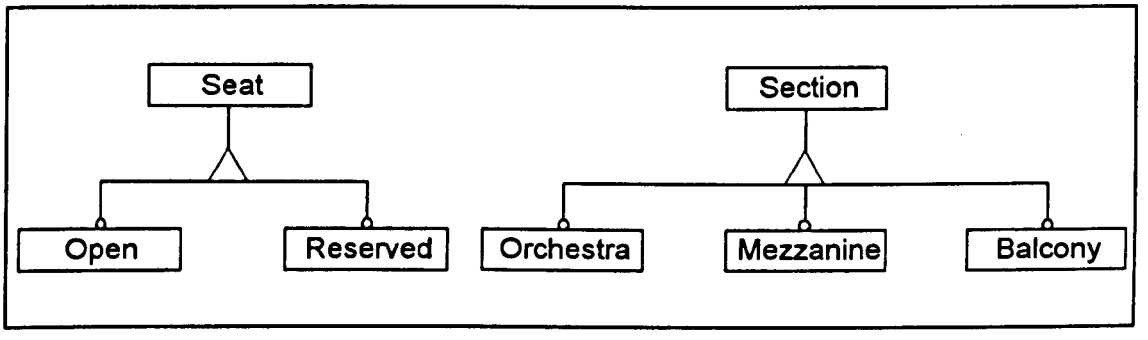

Figure-3 Seating Styles and Section Taxonomy

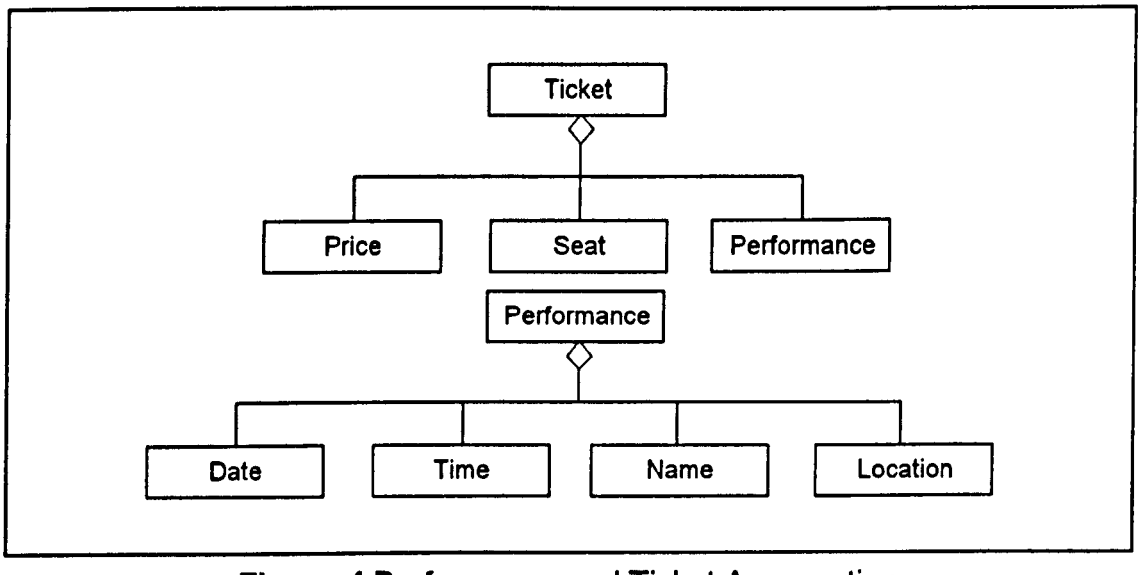

Figure-4 Performance and Ticket Aggregation

Appendix B

Process Control Domain Example

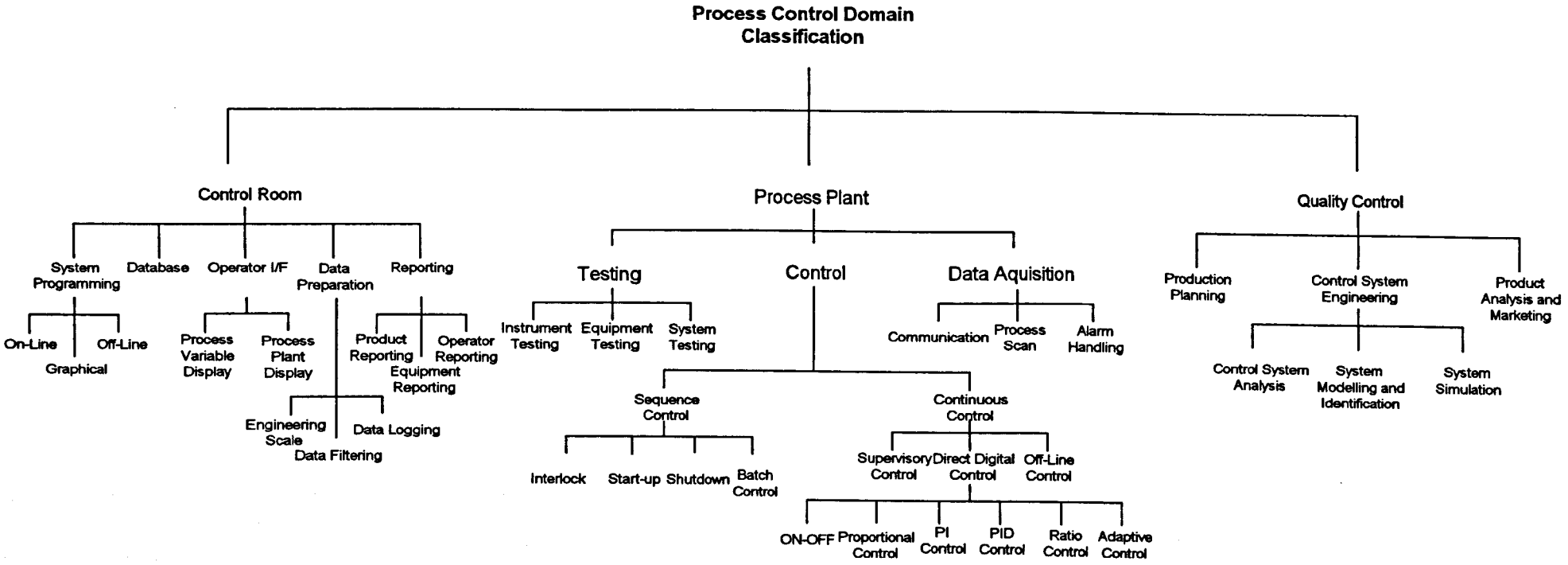

Figure B- 1 Process Control Domain Classification

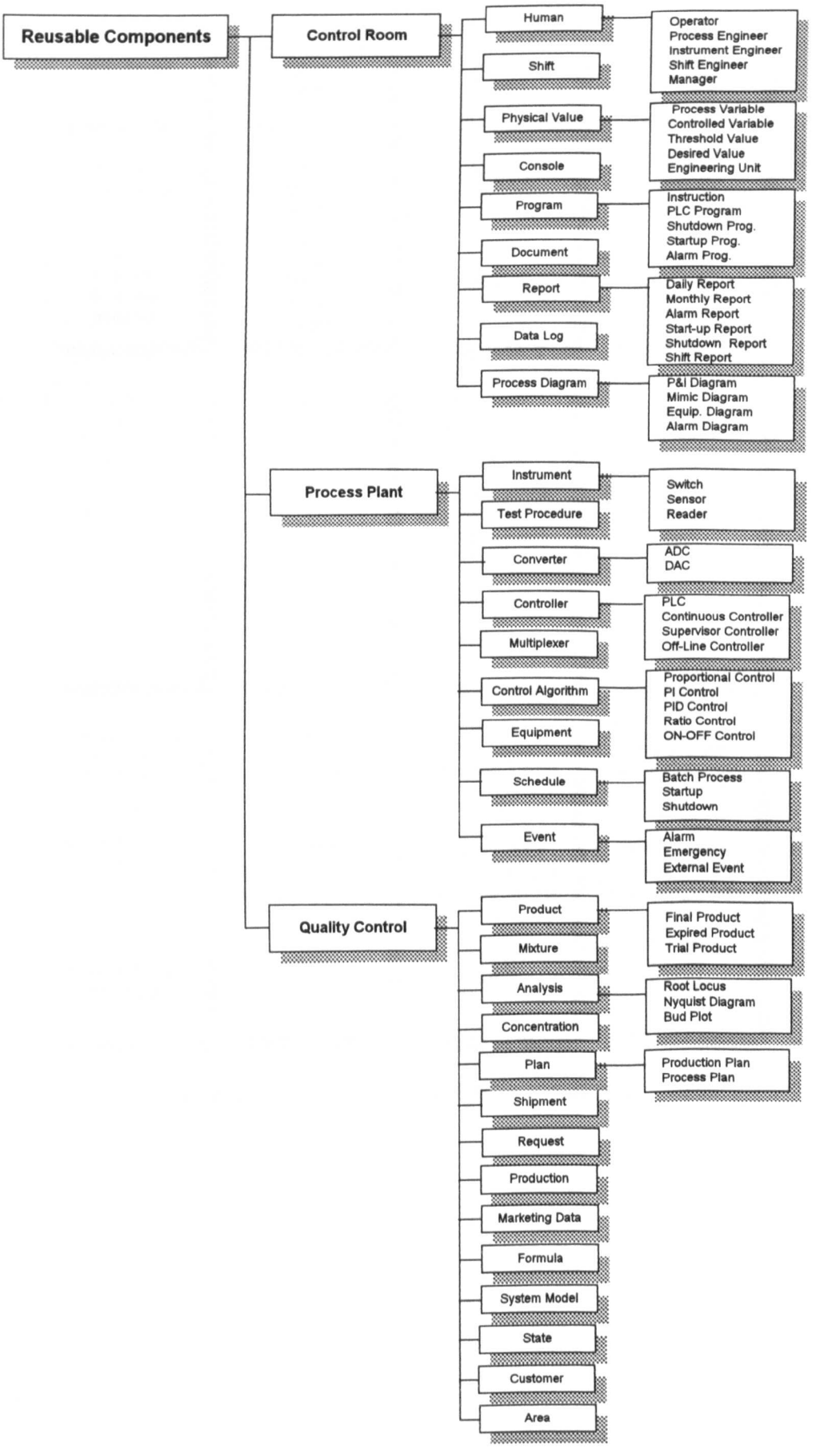

Figure B- 2 Reusable Components Classification

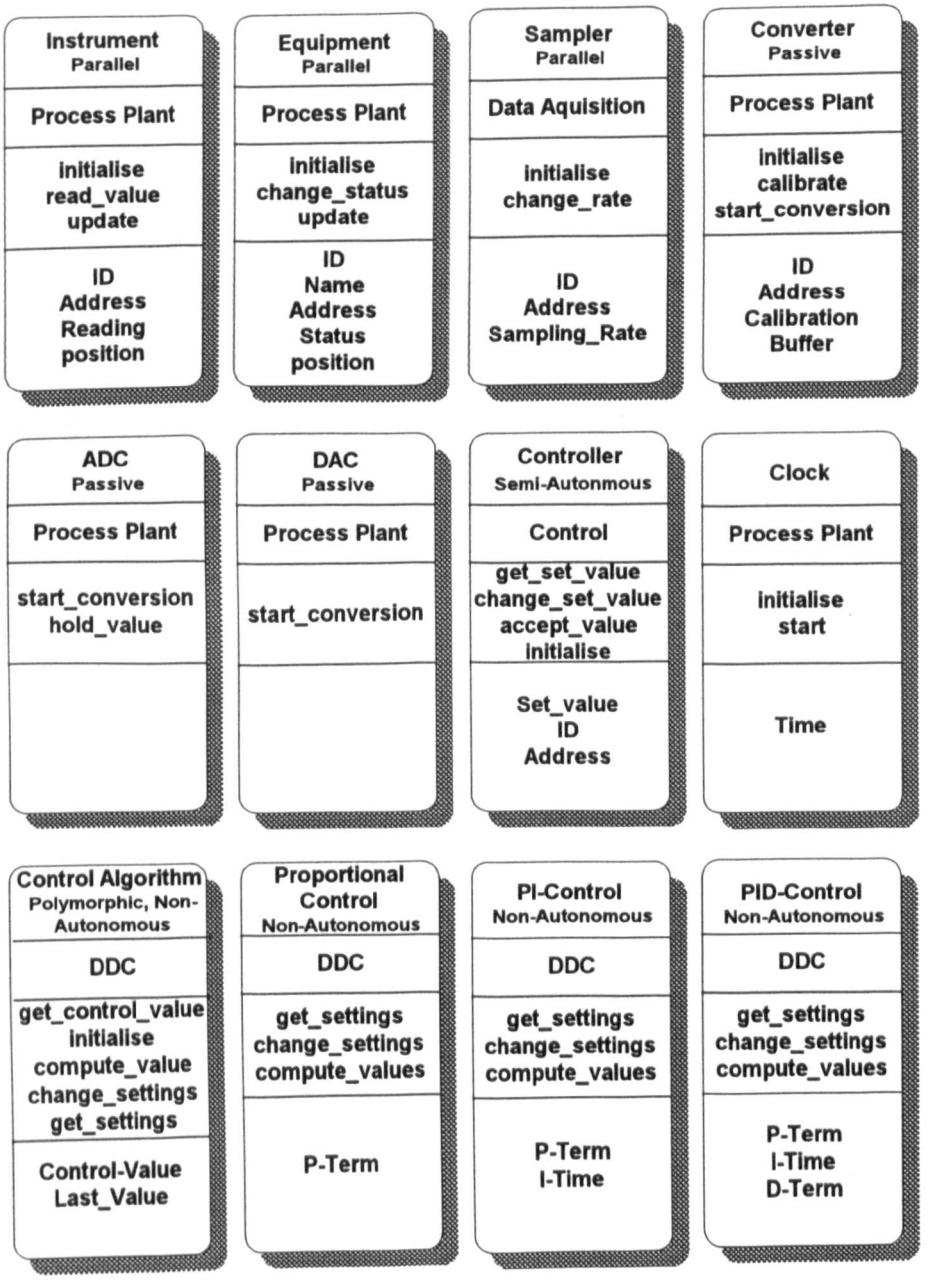

Figure B- 3 Components Specifications