Attribute based component design: Supporting model driven development in CbSE

Kyaw, Phyo

Attribute based Component Design: Supporting Model Driven Development in CbSE

PhD Thesis

Phyo Kyaw

Department of Computer Science
Durham University
Durham DH1 3LE, UK

Abstract

In analysing the evolution of Software Engineering, the scale of the components has increased, the requirements for different domains become complex and a variety of different component frameworks and their associated models have emerged. Many modern component frameworks provide enterprise level facilities and services, such as instance management, and component container support, that allow developers to apply if needed to manage scale and complexity. Although the services provided by these frameworks are common, they have different models and implementation. Accordingly, the main problem is, when developing a component based application using a component framework, the design of the components becomes tightly integrated with the framework implementation and the framework model is embedded in the component functionality, and hence reduces reusability. Another problem arose is, the designers must have in-depth knowledge of the implementation of a component framework to be able to model, design and implement the components and take advantages of the services provided. To address these problems, this research proposes the Attribute based Component Design (AbCD) approach which allows developers to model software using logical and abstract components at the specification level. The components encapsulate the provided functionality, as well as the required services, runtime requirements and interaction models using a set of attributes. These attributes are systemically derived by grouping common features and services from light weight component frameworks and heavy weight component frameworks that are available in the literature. The AbCD approach consists of the AbCD Meta-model, which is an extension of the UML meta-model, and the Component Design Guidelines (CDG) that includes core Component based Software Engineering principles to assist the modelling process for designers. To support the AbCD approach, an implementation has been developed as a set of plug-ins, called the AbCD tool suite, for Eclipse IDE. An evaluation of the AbCD approach is conducted by using the tool suite with two case studies. The first case study focuses on abstraction achieved by the AbCD approach and the second focuses on reusability of the components. The evaluation shows that the artefacts produced using the approach provide an alternative architectural view to the design and help to re-factor the design based on aspects. At the same time the evaluation process identified possible improvements in the AbCD meta-model and the tool suite constructed. This research provides a non-invasive approach for designing component based software using model driven development.

Acknowledgements

Special thanks to my supervisor Dr Liz Burd for all her help and guidance. I also would like to thank my previous supervisor Dr. Cornelia Boldyreff for her encouragement, support and suggestions. My thanks go to Andrew Hatch, Janet Lavery, Sarah Drummond, and Brendan Hodgson and everyone else at the CETL ALiC group at the Computer Science in Durham.

I would like to thank my father Dr. Thein Han, my mother Mrs. Khin Win Yee and my wife, Mi Mon Thet, for all their help and assistance during this research.

Copyright

Declaration

No part of the material provided has previously been submitted by the author for a higher degree in the Durham University or in any other University. All the work presented here is the sole work of the author and no-one else.

List of publications previously made on this research

The following publications have been made based on this research.

Kyaw, P., C. Boldyreff, et al. (2000). Separating Co-ordination from functionality in Component-Based Distributed Systems. Dependable System Middleware and Group Communication (DSMGC2000), Nurnberg, Germany, IEEE.

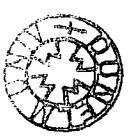
Kyaw, P., C. Boldyreff, et al. (2002). Co-ordinaton Adaptors: The Evolution of Component-Based Distributed Systems. Systems Engineering for Business Process Change. P. Henderson. London, Springer: 298-308.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction



1.1 Background

A successful technology can change the way systems are being developed, fielded and maintained. As an instance, Object Oriented technology has changed the way systems are viewed and composed. However for such a technology or paradigm to be successfully utilised, it must be general and easy enough for users to apply as well as providing facilities for integrating the technology with many existing domains. Commercial organisations are trying to implement or update their systems in a way that such systems or subsystems can be updated incrementally to keep abreast of new technologies and to take advantage of them. These organisations demand not only sound architectures but also efficient ways to integrate different components as well as applying design patterns to solve complex problems that are domain specific.

As Software Engineering matures over time, demands are increasing for software to be developed rapidly with reusable artifacts or assets. Traditionally, in Software Engineering, these artifacts are pieces of code or libraries. However, the term has broadened into representing reusable design, process, patterns, guidelines, frameworks, standards and most importantly components. In other words, contexts of reusability are formed based on different aspects of the artefacts. The concept of aspects in component development will be described in the literature survey. This has led to the total conversion of the perception of the way Software Engineers develop software. Hence new Software Engineering practices have emerged and they have evolved into new Software Engineering disciplines.

This research is about the approaches and techniques to software design based on three emerging disciplines in Software Engineering: Component based Software Engineering (CbSE), Model Driven Development (MDD), and Aspect Oriented Programming (AoP).

1.2 Component based Software Engineering (CbSE)

Component based Software Engineering is a subset of Software Engineering. Like other disciplines in Software Engineering, CbSE aims to provide a mechanism for developing software parts. Unlike other disciplines, however, CbSE is inspired from the building of components in hardware development. Object Orientation (OO) illustrates to software engineers how to model software based on the metaphor of real world objects. CbSE has extended this metaphor to represent software as a set of connected components using a common model and infrastructure. Whilst OO and CbSE share many common concepts such as separating interface from implementation, and encapsulation of internal details, from the CbSE point of view, components can be designed and written in OO or any other procedural languages. In other words, components in CbSE are more loosely coupled and provide functionality as services using interfaces. More detailed description on CbSE is made in Section 2.2.2 as part of the literature survey.

CbSE has matured enough to form various frameworks and their supporting technologies from various researchers, commercial organisations and the open-source community. Therefore it can be regarded as becoming an established discipline. The success of CbSE depends on sound practices, methods, models and guidelines applied by software developers. Currently, however, there is no clear and repeatable practice with a well defined framework when designing components.

Many researchers believe that components and interfaces are the leading way to solve many problems with monolithic applications as discussed in [Vigder and Dean, 1996; Szyperski, 1998]. This research reviews various methods, standards, and technologies in the literature survey and proposes a new method that allows developers to construct components at specification level using model driven approach.

1.3 Model Driven Development (MDD)

Model driven development is another discipline that this research is based on. It can be regarded as a new trend in software engineering. MDD approach is based on concept of constructing models for design. The models can be informal, i.e. on paper or hand drawn, or formal i.e. machine-readable and can be processed. In general, models are used to share knowledge amongst software engineers as well as capturing system design. It is used for specifying, visualising and documenting design artefacts.

With the introduction of Unified Modelling Language (UML) and Model Driven Architecture (MDA) from Object Management Group (OMG) organisation, it is possible to construct new

domain specific models using UML. UML is used and extended in this research, because it is currently a widely accepted modelling language.

To outline the role of MDD, this research does not focus on proposing a new method or practice to improve MDD. However MDD is used for supporting designers to resolve problems with CbSE. In this research, the UML meta-model is extended to construct a new meta-model for component development. Although UML provides class modelling, behaviour modelling, and interaction modelling, the extension in this research is limited to class modelling that captures the static structure of the design.

1.4 Aspect Oriented Programming (AOP)

The aim of this research is to apply concepts derived from Aspect Oriented Programming to CbSE. Partitioning a system into components using CbSE is based on dividing up functional aspects of the system. One area that is not highlighted in CbSE and MDD literature is that of addressing cross-cutting aspects. AOP is concerned with cross-cutting aspects within the design and systemically addresses them – providing another way of organising the design. As the AOP concepts mature over time, there are different frameworks in the literature that allow designers to implement cross-cutting concerns. AOP concepts are applied in this research when deriving a new approach, which allows developers to construct components with abstraction.

1.5 Research objectives and the approach

This section introduces the main objectives of this research. CbSE promised a great deal of benefits for applying the practice. However it needs to be applied using a disciplined practice and there are many problems to be overcome. This research intends to address and improve two problems in CbSE.

Firstly, one critical aspect of a component based development is the requirement for a framework for components be deployed and interacted with. As described by Garlan [Garlan, Allen et al., 1994], without a common framework there may be problems when integrating and reusing components. The component frameworks act as a vehicle for components and takes the responsibility of component management. More importantly, it dictates how the component interacts, using an interaction model. Therefore the designers must have comprehensive knowledge of the particular framework that the design is based on. Accordingly, the architecture of the system design is also dictated by the model supported by the framework. Moreover the component implementation often differs from initial design. This has led to the position that the component is hard to re-use. For this problem, this

research aims to find a way of specifying logical components that encapsulate not only the functional behaviour of the component, but also the properties required by the component when deploying to the component framework environment. In other words, the logical components provide a higher level of abstraction to accommodate evolving component technologies. This is achieved by allowing designers to construct logical and abstract components using a model driven approach. A new meta-model that extends the UML meta-model is formed that tailors the construction of logical components.

Secondly, component frameworks and their associated technologies provide a variety of services such as transaction management, instance management and logging. Most of these services can be regarded as cross-cutting aspects in the design, because they are needed by different parts of the system. However there is no modelling approach that allows designers to identify these aspects and apply them in an abstract way. This research aims to address this issue by allowing designers to define components by explicitly declaring their cross-cutting concerns and forming a composition pattern to apply them.

1.6 Criteria for success

The overall criteria for the research can be measured by evaluating the impact on the artefacts produced during the construction of the components as well as analysing the improvement in the design process of software development.

This can be broken down into a number of different areas that this research aims to contribute in CbSE community. Therefore the criteria are:-

1. Identification of the key factors that improve the quality of design using the core CbSE principles.

2. Development of a new meta-model that resolves the problem of component abstraction and allows designers to construct abstract and logical components at specification level.

3. Development of the component dependency view that highlights the cross-cutting and non-functional aspects of the design.

4. Development of a tool suite that supports the meta-model. This should allow designers to apply the meta-model and enables the component dependency view.

5. Analysis of the productivity of the designers during the development process.

6. Quantitative and qualitative evaluation of the success of the approach based on the case studies.

7. Assessment of the rich set of semantics identified by the meta-model to support the design of component based software systems.

This thesis presents how these 7 criteria are achieved in various chapters.

1.7 Thesis Overview

This thesis contains nine chapters and this is the first. Chapter 2 presents the literature survey where important concepts of the CbSE are discussed. Furthermore, approaches of different component frameworks and their associated technologies are also addressed.

Chapter 3 provides an overview of the model driven development and the use of UML in modelling software components. It also discusses ways to extend UML to apply to other domain specific models.

Chapter 4 introduces the Attribute based Component Design (AbCD) approach and the Component Design Guidelines (CDG) to support the AbCD approach.

Chapter 5 presents the detailed specification of the AbCD meta-model. This includes the detailed description of the attributes defined to encapsulate the component requirements.

Chapter 6 describes the implementation of the AbCD tool suite using the Eclipse IDE. It shows how the tool suite realises the AbCD approach and presents how the component dependency view can be generated.

Chapter 7 describes the two case studies for assessment of the AbCD approach and to allow the evaluation process. The two case studies have two different focus areas of CbSE.

Chapter 8 presents the detailed evaluation process that assesses the impact on the design of the component based software against the aims of the research and further work.

Chapter 9 concludes the research and presents a reflective outline of the contribution of the research.

Chapter 2 Current Research

2.1 Introduction

This chapter presents the background research area and the context surrounding the method described in this thesis. In order to develop a suitable method for designing software components, it is necessary to have a clear understanding of several research areas. This chapter addresses component models and methods of the Component-based Software Engineering (CbSE), and highlights problem areas when developing component-based systems. It also discusses Aspect-oriented Programming (AOP) and Model Driven Architecture (MDA) in relation with the context surrounding this thesis. While the current literature of CbSE is heading towards integration and composition of components when implementing, this research focuses on modelling of components when designing the component-based systems.

2.2 Component-based Development

Component based development using CbSE has significant impact on the software being designed and developed. It has a different process of software development in comparison to traditional software development. While traditional software development aims to develop software by using analysis, design, implementation and testing processes, the component based development approach applies processes of analysis, components acquisition, integration, assembling and test processes.

It is also different from Object Oriented software development in terms of the way components are organised. In contrast to the features of a component, an '*object*' from Object Oriented programming may not be independent, although it also encapsulates its state and behaviour [Jacobson, 1993]. An object can be instantiated and deployed using class templates. It has its own state and identity. Depending on the design of the object, it can be temporary or can persistent. Objects may be instantiated as many lightweight units, which are in contrast to components which may be a heavyweight unit with a single instance within the system.

Once an object is instantiated using a class or prototype object, it requires initial state, which can be set when initialising the object. In either case, an object can be initialised by a static method known as a *constructor* or through an *object factory* design pattern [Gamma, Helm et al., 1995] which is an object itself. A component may or may not be implemented using the OO language.

2.2.1 Background of Software Components

The idea of software components is also derived from computer hardware chips or Lego blocks that allow the software developer to plug in components to the system or allow these components to be composed to form a system. Therefore the chips on the circuit board correspond to the components, and the board itself corresponds to the infrastructure of the component software that includes the architecture, technology and component model that glues the components together, as shown in Figure 1.

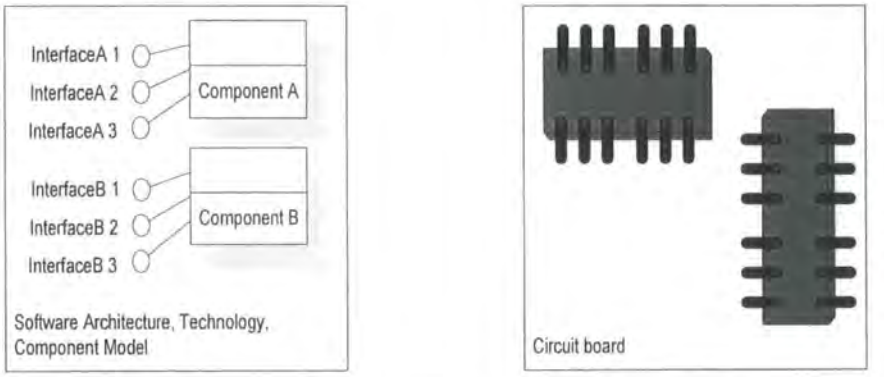


Figure 1 Similar concept of Software Component and Integrated Circuit (IC)

However, the nature of software is different from other forms of hardware products. To create a software component, one has to apply different models and supporting technologies. Although the hardware technology analogy can be used as the basis for component software, the design and development of components require the use of the principles of component architectures as described in Section 2.2.2.1.

2.2.2 The term 'Software Components'

Many researchers have defined the term 'components' in different ways, which can be found in [Jacobson, 1993; Nierstrasz and Tsichritzis, 1995; Orfali, Harkey et al., 1996; Szyperski, 1998]. When reviewing the terms, it was found to be described differently depending on different contexts as discussed in [Caldiera and Basili, 1991]. A component may perform one or more functions depending on the design of the component. The developers can design software components in many different ways. A component can be designed to be used in a custom-based system against a specific interface and architecture. Custom-based systems are systems that are built in-house to fit with the required specification. In this case, the component has to be designed and implemented to fit with the particular system. A component can expect what resources are available on other components and the system. These types of components are mostly a single instance within a system rather than multiple instances of the same component. For example, an Internet server can be a component in a

large information system, which is the single instance within the system. Another type of component is independently deployable component for third-party users, known as Commercial Off-The Shelf COTS [Vigder and Dean, 1996]. In such case, the component can be deployed in a single instance or multiple instances depending on the functionality of the component. Since the component has to be independently deployable, it has to encapsulate all the necessary objects and libraries, so that it can be composed and decomposed easily from its environment and other components.

From the component users' point of view, to compose different components from different vendors, it is very important that each component can be efficiently integrated. From the component developers' point of view, component interfaces have to be completely co-ordinated and clearly defined. Therefore components can be sold without any computability problems [Clements, 1996]. Furthermore, component interfaces should be able to modify easily without having to change the internal structure of the component when integrating with the system [Sametinger, 1997]. A component may be designed and implemented based on an object-oriented approach or other approaches using procedural languages. However, unlike objects, a component may contain classes, objects or other functional modules and procedures. A component can also apply functions and procedures from other components and use their resources. The functionality and the properties (i.e. classes, objects, and modules) should be clearly defined within the component.

Many researchers have defined the term software components in different ways. Traditionally, any unit within a software system, such as subsystems, procedures, modules, objects, can be regarded as different components of the system [Clements, 1996]. However, as the software engineering evolves over time, the terminology has also changed. The term software component that is used in CbSE is different from the general components within the system. The term software component is a unit which its internal implementation is hidden as a black box and can communicate only through its well defined interfaces [Szyperski, 1998]. A component is not an object from the Object Oriented programming [Rumbaugh, Blaha et al., 1991]. A component can be implemented as an object or a collection of objects, functional procedures or even a set of libraries. However once the component is implemented, the clients that use the component do not need to be concerned with the internal structure of the component and only have to deal with its one or more interfaces exposed by the component.

As the foundation of software engineering has matured over several decades, most developers have reused many existing designs and code, and it is rarely the case that they start from scratch [Meyer, 1994]. However, software components that are able to be plugged into the system and used as necessary by an end-user are far from being viable because at the moment there is neither a successful component market nor a large number of development

communities. Amongst many different aspects of the component-based software to be researched into, this thesis focuses on component modelling, component technologies, non-functional aspects of enterprise components and abstractions in component development.

However from this research point of view the term software component can be defined as '*a software unit or a building block, which can be independently deployable and composable with other software components, permitting that component contracts are satisfied, and component framework are compatible, to form a component-based system*'.

It is defined here to narrow the scope of the research and to define principles of software components that are focused in the thesis. More specifically, this thesis focuses development on small to medium size enterprise level software components that are deployable components as models rather than technology specific implemented components.

2.2.2.1 Principles of Software Components

The definition presented in the previous section is based on the principles of Software Components. Each part of the definition can then be divided to elaborate the meaning and to provide the clear understanding of the principles.

- **software unit or a building block**: Each software component can be regarded as a '*self contained*' software unit. In other words, it not only encapsulates its implementation details but it is also composable with other components using '*well defined interfaces*'.
- **independently deployable**: As a component is sufficiently self contained, the changes made internally (i.e. to its design or implementation) do not affect other components as long as the interface contract remains intact.
- **contracts**: For a component to be independently deployable and to be self contained, it must have well defined interface that can form a contact on what it can provide and require to function with other components.
- **component architectures**: To be able to form a component based system, all components must be based on a common component architecture, which includes the specification (and/or implementation) that describes the component model, the wiring standard and the framework.

Defining these principles and the definition of the term software component has raised the question of whether the term can be broadened to a design perspective. Currently, the component as a software unit or a building block is referred to as implemented code, or a module which is at the implementation level rather than a specification block at the design

level. In the literature, there is no clear definition for the specification component at a higher abstraction. This research focuses on components at the design level to gain more abstraction over implementation.

2.2.3 Properties of Software Components

The following sections focus on the properties of Software Components. After describing the concept of Software Components and the context surrounding the term, the properties of the Software Component are described here for the understanding of the rationale for using component based approach to construct enterprise-level software.

2.2.3.1 Component Interfaces

The component interface is the most important aspect during component composition. An interface can be regarded as the entry gate to the component. It consists of all the services and functions provided by the component to be used by other components and the system itself [Cicalese and Rotenstreich, 1999]. The *contracts* can be used as common specifications for interfaces during component composition [Hondt, Lucas et al., 1997; Beugnard, Jezequel et al., 1999]. The component providers can implement interfaces according to the contracts and users can use the interface specifications that are stated in the contracts. Accordingly, the usefulness of a component not only depends on the functionality but also on the interfaces that the component provides, including its portability, extendibility and adaptability.

The Interface Definition Language (IDL) has emerged to describe the services provided by a component [OMGIDL]. An IDL file contains one or more interface definitions. Each interface describes operations, parameters to these operations, and data types. However the components require facilities for describing required interfaces as well as provided interfaces which is not supported by the initial development of IDL [Olafsson and Bryan, 1996; Canal, Fuentes et al., 2003]. The CORBA Component Model (CCM), which is part of the CORBA 3.0 specification, introduced new additional features to IDL by including facilities to specify 'required services' as well as 'provided services' as well as events [(OMG), 1999]. The notion of such features is defined using the term facets, receptacles, event sources, and event sinks which are described in [(OMG), 1999]. There is also another version of IDL introduced in the COM and COM+ framework by Microsoft [Rogerson, 1997]. The detail of each approach is discussed in Section 2.2.4 when presenting component frameworks.

2.2.3.2 *Contracts*

A component can communicate with its clients using the interface. Contracts are used as interface specifications between the client and the service provider. In other words, contracts can be used as rules that the clients and providers could agree on for efficiency. One of the most important contracts between interacting clients and providers is setting '*pre-conditions*', '*post-conditions*' and '*invariants*'. The pre-conditions can be set before any operation is carried out and post-conditions can also be set to ensure that the operation meets the required conditions and results [Meyer, 1994]. With this specification, the component providers can change or update their component implementation without changing the interface, which can make the component independently deployable. Therefore both existing clients and new clients can use the new version of components. However the pre-conditions and post-conditions are not the only way of specifying contracts. Other ways include the use of '*non-functional*' specifications on reliability, response time, performance, independence and security, and can be set to minimise risk. As the logic and techniques become more complex over time, the contracts become more and more complex between clients and component providers. A client may involve a call to a component service, which needs constant feedback from the component to the client to indicate the state of the progress throughout the operation. In such cases the client and provider need to synchronously communicate during the operation and therefore pre-condition and post-condition specifications are not effective. A callback can be a procedure or an object that passes to the service provider (i.e. a library or a remote service) which can call back the client for information [Beugnard, Jezequel et al., 1999]. When the *callback* mechanism is introduced when calling from the client to the provider, the contract may become invalid. This is because the pre and post conditions of the provider set at the beginning of the call may be changed during the process. Therefore, when the call back is made during the intermediate process, the observable state of the provider may be different from the initial contract defined at the beginning of the call. Accordingly the client must aware of the call back state of the provider and should be not dependent on the intermediate state of the provider.

In the object oriented community, the Object Constraint Language (OCL) was introduced as part of the Unified Modelling Language (UML) to describe constraints on object artefacts modelled using UML [Warmer and Kleppe, 2003]. OCL is a formal and expression-based language, and can be used to express more precise and unambiguous specification. Users of the UML and other modelling languages can use OCL to define contracts by specifying constraints and other expressions. OCL is intended to add constraints to operations and properties of the objects.

For example, to specify the maximum number of passengers on a Bus class:-

```
Context Bus
Inv : bus.NoOfPassengers <= 30
```

From the software components point of view, as the OCL is intended to add constraints to objects, it can only be used as a basic tool to specify functional requirements of a component interface. More concrete specifications and standards are required to facilitate software components when adding non-functional constraints to form more efficient contracts. While OCL is intended to complement UML models by allowing the addition of ***constraints*** at the design level, at the implementation level, Object Oriented programming languages such as Eiffel which includes build-in features to add constraints to classes using the concept of '*design by contract*' [Meyer, 2000].

The use of contracts is vital to the success of component integration in any software development. The above discussion shows that the use of OCL and other languages focuses on conditions and specification statements of functional aspects. However as one of the fundamental concepts behind the use of software components is to be independently deployable, contractually-specified interfaces play a major role during component composition. The contract should cover not only the functional aspects but also non-functional (or extra-functional) aspects such as component's **performance, availability, persistence state and security**. In the literature, achieving such features is yet to be explored and researched. This research explores different ways of specifying context based constraints to allow the developer for adding add ***non-functional contracts*** when constructing software component artefacts.

2.2.3.3 Patterns

Solutions to common software design problems are recognised as software engineering evolves over time and become '*design patterns*'. Some researchers have formally defined the term 'design patterns' as:-

> "Design patterns are recurring solutions to design problems you see over" [Alpert, Brown et al., 1998]

> "Patterns identify and specify abstractions that are above the level of single class and instances, or of components" [Gamma, Helm et al., 1993]

Accordingly, each design pattern consists of the problem domain or the context, the problem itself, and one or more solutions to the problem. A collection of design patterns was

documented by Helm and Erich Gamma in their Gang-of-Four book [Gamma, Helm et al., 1995]. Design patterns were originally incorporated from the domain of GUI application framework ET++. As the design patterns become more common, the patterns community has proposed many different types of patterns such as architectural patterns and design patterns as described in [Schmidt, Stal et al., 2001]. However the nature of the patterns in different domains can be varied. Therefore some of the patterns are domain specified and have limited application. To solve more complex and recurring problems, a single pattern solution is inadequate. Accordingly, the community and researchers have proposed '*compound patterns*' to combine different smaller patterns [Riehle, 1997]. These combined patterns are then derived to form '*pattern languages*' as presented in [Alexander, Ishikawa et al., 1977; Martin, Riehle et al., 1998]. From the CbSE point of view, components are glued together using the architecture the components are based on. A design pattern can be regard as a micro-architecture as described by Szyperski as "Design patterns are microarchitectures. They describe the abstract interaction between objects collaborating to solve a particular problem." [Szyperski, 1998]. Modern component technologies such as J2EE, COM+ and CCM use various patterns which provide guidelines to ensure that components are integrated efficiently within the boundaries of the component architecture. As an example, J2EE utilises several patterns including [Crawford and Kaplan, 2003]:-

- Façade/proxy pattern: for handling synchronous communication and remote operations.
- Publish/subscribe pattern: for decupling component service providers from service consumers.
- Factory pattern: for separating the management of objects and objects activation.

There are also 'antipatterns', which are similar to design patterns but are formed when developers make mistakes when trying to solve common and recurring problems and the solutions to correct the mistakes. In other words, antipatterns are recurring mistakes, and design patterns are recurring solutions [Brown, Malveau et al., 1998].

The study of design patterns is important to this research, because in the context of CbSE, full comprehension of design patterns give more understanding of the component architectures that can be formed.

2.2.4 Component frameworks, standards and technologies

This section presents a survey and evaluation of different component frameworks, their standards, models and technologies that support them. In particular, from the three most dominant players of the emerging component market, i.e. COM/COM+/.NET from Microsoft,

which is derived from component framework for building desktop and GUI components, CORBA/CCM from OMG, which is originated from enterprise level distributed computing and remote architectures, and J2EE from Sun Microsystems, which the technology is centered around the features of Java programming language and Web Server-based components. The study of different component frameworks is important for this research. Most of the studies in the literature provide detailed features provided by each framework. This study tries to find common facilities and services provided by all frameworks to gain an abstract view and to find a way of form a logical component framework.

2.2.4.1 Component Object Model (COM), COM+ and .NET

The Component Object Model (COM) is one of the component oriented frameworks from Microsoft. COM consists of a specification for constructing components and partly an implementation in the form of a standard API. Although COM has similarities to CORBA, COM is based on a different approach as described in [Rogerson, 1997]. COM is targeted for Microsoft Windows environments although Microsoft is developing for other platforms by third parties, such as SUN OS, Macintosh, HP, etc.

COM is the basic foundation of all Microsoft OLE, ActiveX, ActiveX Data Objects, and Automation controls technologies. The COM component framework is studied in this literature to gain better understanding of how this technology implements the core CbSE concepts. Hence the mapping between components at design level and COM components can be constructed to provide an abstraction from at the Design level.

2.2.4.1.1 Component Model

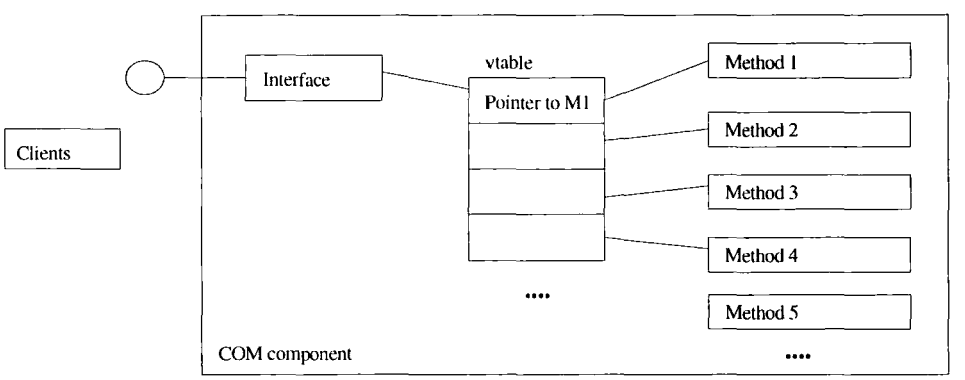


Figure 2 Component interface and vtable of COM Model

COM components are 'black box' binary form of units which use *Interfaces* to communicate with the outside world. A COM component consists of one or more interface nodes. The available services are exported using interfaces that the clients can reference to. Therefore an

interface can group a set of related properties and functions. As shown in Figure 2, this grouping is arranged as a table of pointers to those functions, which are called virtual functions (vtables).

Hence each interface has a separate function table. The client can point to the interface pointer that points to the first field of the interface node and then the interface pointer points to the function pointers that point to the various functions available. For instance, the interface node Icalculate may contain four member functions called add, sub, mul, and div. The function table contains the addresses of GUIDs (Globally Unique Identifier) for those four functions.

Therefore if the client wants to make a call to the sum function, the interface pointer can dereference to the sum function pointer and then to the sum function itself. As a summary, an interface is a pointer to a function table which is a list of pointers that point to associated functions. COM also supports programming languages that do not provide pointers such as Java and Visual Basic. COM uses the special interface called Idispatch which are names referring to the related functions. This is called *dynamic invocation*. These names use a standard data type (variant) to allow clients with different programming languages. Since the type variant (16 bit packet) has a fixed size, it can be passed as parameters for dynamically invoked functions.

These interfaces can map to one or more classes within the component. Hence, the implementation can be made freely depending on the functionality and design of the component. A COM component may consist of one or more objects which provide different services. There can be also modules that are not within objects. COM uses the concept of a Globally Unique Identifier (GUID) to identify a component and its interfaces. A GUID is a 128-bit number which guarantees to be unique for all components and its interfaces. Hence, components use Class Identifiers (CLSIDs) and Interface Identifiers (IIDs) as unique identifiers. By giving the CLSID to the COM API function **CoCreateInstanceEx**, a instance can be created and loaded for the clients to use.

All COM components must have IUnknown interface and all interfaces are derived from IUnknown interface. The IUnknwon interface has the three main function methods. They are **QueryInterface**, **AddRef**, and **Release** methods. The QueryInterface is in every interface since all the interfaces are derived from IUnknown. It allows the clients to query available interfaces (i.e. services) provided by the owner component. Once the client retrieves a particular interface, the reference counter is increased by using the AddRef method. It decreases the reference counter by using Release. When the reference counter turns to 0 the component

unloads itself since there is no client using the service. Therefore every component performs *reference counting* for the whole component or an individual interface.

All the information and services provided by a particular component is presented in its *type libraries*. Type libraries provide all information about the component's interfaces where the developers can learn about the components. Type libraries can be created by writing scripts in the Microsoft Interface Definition Language (IDL) or the Object Description Language (ODL) and compiled using a compiler. However Microsoft's IDL does not conform to the standard OMG IDL and therefore it is not standardised with other language independent protocols.

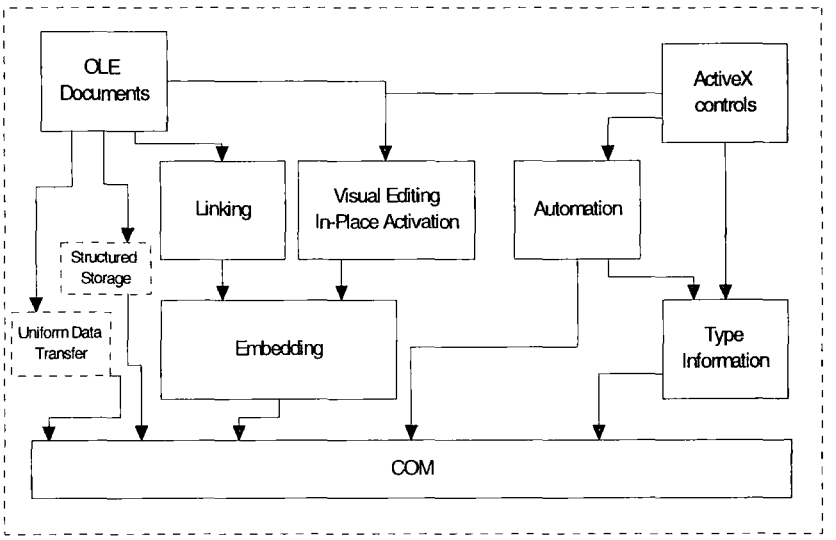


Figure 3 COM model overview

Components can reside within DLL libraries, EXE executable files or OCX (ActiveX extensions) component servers. All components can be registered to Windows system registry with its CLSIDs and actual location of the servers. Hence once the client requests a particular component, the above function looks up the registry to locate the location of the component server. Once the server is located, the server uses *class factories* to retrieve the requested components. A class factory is the special type of interface object attached to each component within the server. There is also *class factory 2* which needs additional licensing to create its component.

A COM component can provide its service to all *in-process* clients and *out-process* clients depending on the design of the component. In-process clients are the clients within the same process. Out-process clients may call from different process on the same machine or from the remote machine. In both cases, client and server do not know if they are making calls to the

remote or local machine, as it is all performed by a client side proxy object and server side stub object. These two objects perform marshalling between the two components

For a local machine, the proxy sends the Interface pointer identifier (IPID) and Object identifier (OID) to the server stub to locate the particular interface object and to locate the local proxy object for returning the request. For clients from a remote machine, additional information is added within the *object exporter.* This information is represented in *network data representation* (NDR). This whole communication process is performed by Distributed Component Object Model (DCOM).

As the COM specification evolves over time, Microsoft has released many extensions. Figure 3 COM model overview

shows the complete COM model and its extensions.

Object Linking and Embedding (OLE): OLE technology includes a collection of COM services such as drag-and-drop controls, monikers, connectable objects and automation support as discussed in [Chappell, 1996]. OLE also comes with different extensions such as OLE containers and servers, ActiveX controls and ActiveX documents. OLE controls usually have visual appearance and are most suitable for document-centric applications. However developers are allowed to use OLE technology within Microsoft frameworks or build their own framework to develop the component. Hence the technology lacks openness and is limited to open binding and linking of components.

Automation: Automation is also a COM based extension that allows the application to control the objects in one or more applications, like macros. The client is refereed to as the automation controller and the server is called the automation server. Automation can be performed in-process, local or remotely. The exported functions for scripting can be found in type libraries of server objects. Therefore automation is suitable for building scripting applications or for the automation of services. However COM automation does not comply with the OpenDoc scripting technology.

ActiveX controls: ActiveX controls are another extension of COM components. They are mainly in visual form and can be embedded only to in-process servers and ActiveX container applications. ActiveX controls have exported visual related services such as input events, data sources and licensing.

2.2.4.1.2 Reuse

COM supports code reuse by providing COM servers in the form of dynamic link libraries (DLLs), COM based executable (EXE) and OCX servers. Furthermore, Microsoft has provided a set of reusable APIs such as Win32 API and Win32 SDK which contains a collection of documentation and header files that allows the developers to reuse Windows System DLLs.

2.2.4.1.3 Language Independence

Since COM provides dual interfaces, development can be made on programming languages that support pointers such as object pascal, C++ as well as languages that do not directly support pointers such as Visual Basic and Java. However these languages must be COM compliant and must use COM supported data types. Furthermore, extra dual interfaces have to be implemented for programming languages that do not support pointers such as Visual Basic. Therefore implementing COM components using programming languages that support pointers such as C, C++ can be easier.

2.2.4.1.4 Portability

Although Microsoft claims that COM support on other platforms is currently developing, at the moment the COM specification is limited to only Microsoft windows environment. Furthermore, the use of Windows Registry is also limited to Microsoft Windows based operating systems.

2.2.4.1.5 Object Memory Management

COM components provide self creation by using class factories and self destruction by using reference counting to manage memory and to manage object lifetime. However there is a lot memory overhead and performance problems for distributed components when communicating each other because the architecture involves overhead objects.

2.2.4.1.6 Object Orientation

COM provides two forms of object orientation. Firstly, COM is the binary standard and it offers encapsulation of objects. In instance, the client does not need to know how the server object has been implemented and the server does not know where and how the client is calling. Secondly, the developer can create COM objects in such a way that they can be updated or substituted at run-time, which is the concept of polymorphism. Therefore, objects

can be replaced without interfering with the interfaces, accordingly the clients do not need to recompile for any changes.

2.2.4.1.7 Distributed Services

DCOM provides facilities for implementing distributed systems by using client proxies, server stubs and marshalling methods. However, unlike CORBA, the DCOM distributed technology is only available in Micorsoft's world and there is a lack of support for other third party platforms.

2.2.4.1.8 The summary of COM model, COM+ and .NET

To summarise the COM technology, it was developed to satisfy some of the core CbSE principles.

Firstly, COM provides the separation of interface from implementation. This makes the client of the component independent from component implementation. COM provides Type Libraries for publishing interfaces as well as supported types by the component to be discovered by the clients. The 'Binary compatibility' is the core principle of the CbSE to enforce the separation of the client from component implementation. This ensures that the client does not need to recompile or redeploy when updating or changing component implementation as long as the component provides the same interface. COM provides this by creating the indirection between the methods that implement component functions and the client using interface pointers as described in COM component model, in Section 2.2.4.1.1. However, once the interface is published, the pointers are fixed to the memory addresses of the implementation methods and hence changes to the interface will break the contract between client and the component. That is the single most shortcoming of the COM model from CbSE point of view. .NET framework, which is the successor of the COM framework, provides alternative approach for binary compatibility. It uses attributes as meta-data of the component to expose its methods and fields. Unlike COM the memory address of these methods are link by JIT compiler at runtime when the client invoke the methods. This is similar to JavaBean component model where it relies on Java JIT compiler to provide the linkage.

Secondly, COM provides packaging and deployment of components with versioning support. The shared components can be packaged into DDL. However, it needs to be installed to the system for the client to use the component. Therefore extra care is needed for the developer when updating the component to a new version, because clients are fixed to a particular

version of the component. This is because of the static interface linkage between client and the component.

Thirdly, COM also provides other services and tools for construction and composition of COM components. Security services, directory services, transaction services, licensing services, Object pooling, Just-in-Time Activation are some of the services provided by the COM specification. It uses COM+ component services, which is the COM component container to provide these services. COM+ provides a runtime environment for components to be deployed on.

As an overview, although OMG's CORBA and JAVA Bean technology are technically better (i.e. much more open), the COM technology is in control of the of today's desktop applications. With the background of Microsoft Windows, many organisations have developed and used COM models and invested in COM based technology.

2.2.4.1.9 Development Steps using COM

The development steps here are focused only on COM oriented component implementation stages and not intended to discuss general component-based architectures and frameworks.

One of the most advantageous features for using COM is that there are many easy to use tools such as MS Visual Studio and Borland Delphi which uses wizards. Accordingly, the development steps are varied depending on the tool used and types of application or component to be developed. However in general the development involves the following steps.

- Design and selection of type of applications or components to be developed. (i.e. automation servers, automation controllers, type libraries, ActiveX controls, and other visual components).
- The designer has to decide the COM components as in-process, out-process or remote servers components. This defines whether the components are shared or private to clients.
- Design and construct a set of interfaces for components and services for the server.
- Based on the design the developer has to design which threading model to use, and whether to include type libraries for more information about the component.
- Implements the components and depending on the types components can register within EXEs or DLLs or OCX servers.

For registering multiple components over the internet, the developer may use tools such as Microsoft Transaction server provided by Microsoft.

2.2.4.1.10 Application Domain

Most of the applications produced by Microsoft are based on the COM model. Accordingly, Microsoft is dominating the market for many industries and organisations with its operating system, Windows, and many other applications based on COM. One of the most popular components that are based on COM is ActiveX components which are derived from the earlier OLE control technology. However, ActiveX technology is most suitable for small and lightweight visual components. Presently, many of the current technologies such as DCOM, Automation servers, Microsoft Transaction servers, DirectX, ADO, etc are based on COM component model.

2.2.4.1.11 Development tools

Most of the current available tools by Microsoft and other third party tools support the development of COM components, ActiveX controls, Automation servers, and component libraries. Some of the development tools available are Microsoft Visual Studio, and Borland Delphi. This is one of the biggest advantages of COM in comparison with other architectures - these tools are easy to use and powerful. However the limitation is that the applications produced by these tools are platform dependent. COM components can also be implemented using tools such as J++ and Active Perl, with some limitations.

2.2.4.2 CORBA

Since Object Management Group (OMG) was first founded in 1989, they have first introduced the Common Object Request Broker Architecture (CORBA) to overcome component integration problems. The CORBA 1.1 specification was first established in 1991 and followed by the 2.0 specification in 1995. [OMG, 1997] The main feature of CORBA includes the integration of components with language, location, and platform independence. It allows different vendors construct different components and integrate them. One of the most important specifications of the CORBA is Internet – inter – ORB protocol (IIOP) that is specified in CORBA 2.0. Any vendor which wants to make software compatible with ORB must support IIOP. OMG also defines the Object Management Architecture (OMA) which combines with CORBA to form a complete middleware architecture for distributed systems. However OMG defines CORBA as a specification and component implementation framework there are many independent commercial and open source tools available.

2.2.4.2.1 Component Model

The three main features of a CORBA service are a set of invocation interfaces, the object request broker (ORB) and a set of object adapters [Emmerich, 1997]. The invocation interfaces allow late binding. In other words, the method implementing the invoked operation is selected based on the object implementation to which the receiving object's reference refers. Since the components and services are implemented in different languages, there must be a common interface language for communication and integration. The OMG has introduced a common interface language called the Interface Description Language (IDL). Hence, invocation interfaces and object adapters can work together using IDL. When the server wishes to provide a service, the method interface is to be written in IDL. The IDL compiler compiles the IDL and registered in the ORB's interface repository. These available interfaces can be retrieved from the ORB interface, as shown in Figure 4. The interfaces can then be implemented and registered to the ORB's implementation repository. These implemented components or fragments are called object servers. Therefore the implementation can be changed or updated without affecting the interface. When a client wishes to perform a request on a method, the client can use Dynamic Invocation Interface or an OMG's IDL stub. A stub can perform all the marshalling to the remote method though the ORB to the remote server and serve the result as a local object. When the server skeleton receives the requests, the data is unmarshalled and invokes the target method. Once the request is made, the requests are marshalled and sent back to the stubs.

As shown in Figure 4, the ORB core is responsible for locating the appropriate implementation object and transfers control to the object implementation through an IDL skeleton or a dynamic skeleton. Therefore, the object servers can obtain services from the ORB core though the object adapter. Accordingly, based on the type of service required, the object server decides which object adapter to use.

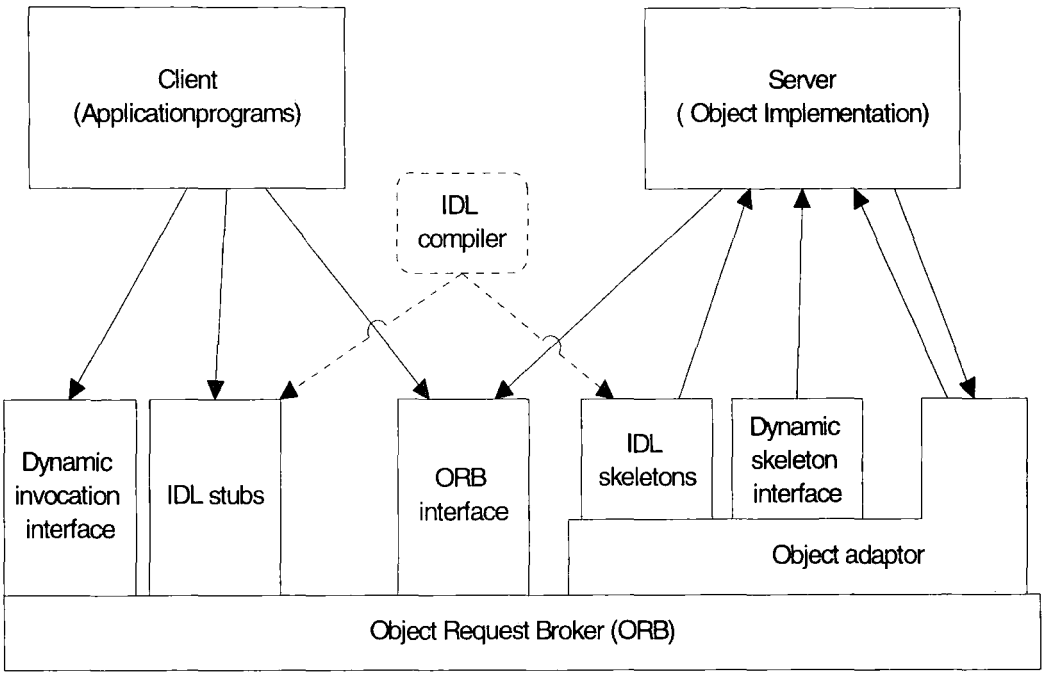


Figure 4 the structure of CORBA's Object Request Broker Interfaces

Object Request Broker (ORB): Object request broker can be implemented in many ways. However, many ORBs include IDL compilers, object repositories, interface repositories, and object adapters. Furthermore, more than one implementation of an ORB can exist with different object references. Starting from the ORB core, additional layers can also be added depending on the services.

For different services and communication, the ORB can be implemented using slightly different styles. These styles include *client and implementation resident*, *server-based*, *system-based*, and *library-based* ORB. For more information about ORBs the reader is referred to CORBA 2.3 specification manual [OMG, 1999].

Dynamic Invocation Interface (DII): Stub routines can be used to perform static binding that is specific to a particular operation on a particular method. To dynamically invoke an operation, or construct an object, dynamic invocation interfaces can be used. In this way, the client can specify which object to use and what types of operation to perform by providing information about parameters and their types. The client may obtain this information from the interface repository.

Dynamic Skeleton Interface (DSI): From the point of view of the server, the objects can be accessed by static skeleton interfaces which map to the methods that implement each type of object, or by an interface which allows dynamic invocation of objects. Dynamic Skeleton Interface provides access to the operation name and parameters in a manner analogous to the client side's DII. [Emmerich, 1997]

Object Adapters: Object adapters can access the services provided by the ORB. The ORB also uses object adapters to provide many interfaces to different kinds of object implementation for providing properties including granularities, lifetimes, policies, implementation styles and others. ORB also provides different services such as security of interactions, object and implementation activation and deactivation, mapping object references to implementations and registration of implementations.

2.2.4.2.2 Development Steps

Development steps can be varied depending on the different attributes of the whole architecture. General development steps are:

- Selection of which CORBA implementation to be used.
- Design of the required architecture based on CORBA specification.
- Design by a top-down approach from general framework to detailed services of each component, or bottom-up approach which begins with services and properties of each component towards the top level framework.
- Creation of components, including required interfaces and implementation.
- Creation of libraries to support different services of the framework.
- Configuration of wiring methods to integrate components.

These steps can be varied depending on the types on application, language used and the implementation tools. Although CORBA provides support for non-object-oriented languages, in general, object oriented languages have an advantage in implementing CORBA applications.

2.2.4.2.3 Application Domain

CORBA is suitable for developing distributed applications in many different areas such as banking, telecommunication, electronic commerce, etc. Since CORBA is platform and language independent and the largest consortium in the software industry, there may be many possible different domains.

2.2.4.2.4 Development tools

Since OMG has proposed CORBA as a specification, many different vendors have implemented the CORBA specification. These include VisiBroker by Borland (formerly by VisiGenic), ORBIX by IONA, PowerBroker by Expersoft, SmallTalkBroker by DNS Technologies, etc. Some of the implementation tools are open-source and some are commercial. Accordingly commercial implementations such as Borland Visibroker (and Borland Application Server) provide more facilities than other free implementations. For more information about comparing different CORBA implementation tools the reader is referred to the CORBA comparison Project report by Distributed Systems Research Group and MLC Systeme GmbH. [GmbH, 1999]. There are also bridge tools which allow CORBA components to communicate with other component architectures such as COM, such as CORBAplus and ActiveX bridge by Expertsoft.

2.2.4.2.5 Summary of the CORBA and CbSE

This research focuses more on component model provided by CORBA specification, and less on the distributed services. To summarise the component model, the study shows that although CORBA specification provide interface based composition of object, it focuses on providing infrastructure for distributed system construction and does not support core CbSE principles. For instance, CORBA specification does not include the component framework, (i.e. the runtime envonrment that manages CORBA components). Presently, OMG has introduced the CORBA Component Model (CCM) specification that addresses the component framework. It allows developers to construct CORBA components using the high level component structure using Interface Definition Lanauge 3. The CCM also defines how the components should be deployed and assembled using XML configuration and property files. Due to the lack of commercial support and implementation, CCM is yet to be regarded as a mainstream component technology.

2.2.4.3 JAVABEANS

After reviewing COM and CORBA, JavaBeans from Sun Microsystems is discussed here. JavaBeans are the collection of Java components developed by Sun. Java Beans are developed mainly for visual programming and can be used by visual programming tool builders. As the model is based on the Java programming language, most of the component construction techniques are based on Java approaches.

2.2.4.3.1 Component Model

Like other component models, JavaBeans also support run time discovery of objects and integration. As mentioned above, JavaBeans are intended to be used within visual application development tools. Most of the bean's services, methods and event handling features are visually oriented. In other words, the properties, methods and events of a bean can be changed using a visual interface by the development tool when developing applications. Like OLE objects, JavaBeans can be embedded and manipulated within standalone applications or applets. However, Enterprise JavaBeans (EJB), discussed later in this section, are mostly non-visual components and reside in the server environment for distributed systems. The following features are some of the most important features provided with JavaBeans.

Persistency: JavaBeans architecture uses the object serialisation techniques to store beans in a persistent state. The persistency is achieved by streaming the object to save and restore the state of the object.

Introspection: The JavaBeans architecture also provides introspection to expose a Bean's properties, methods and events. This can be done by creating the JavaBean Info class. Therefore Bean Info object is separated from the actual bean object. Introspection uses a meta-object to provide information on its behalf. However this Introspection mechanism is mainly useful only for visual development tools [Englander, 1997].

Customisation: A JavaBean is a fully customisable object. Using the customiser, which is a user interface for customising an entire bean, the properties and behaviours of the bean can be fully customised and configured. Another way to configure the property, state and behaviour of the bean is by using *Property Editors*. Property editors can be used by visual development

tools to visual edit the initial or current state of the beans. This customisation can be made both at design time and at run time.

Design Patterns: Many researchers and programmers have suggested different design patterns for various aspects of JavaBean components. This includes patterns for event related objects, listeners, notification, and other proper access methods. An example design pattern for accessing properties is:

```
public void set <PropertyName>(<PropertyType> value);
public <PropertyType> get<PropertyName>();
```

2.2.4.3.2 Enterprise Java Beans (EJB)

Enterprise Java Beans are extensions of Java Beans which can reside on application servers in order to provide different services. These components use the *Remote Method Invocation (RMI)* interface to communicate with their clients. In the future, Sun claims to integrate with CORBA's IIOP and DCOM models, which allows the bean components to integrate with CORBA based components and Micorsoft's ActiveX controls.

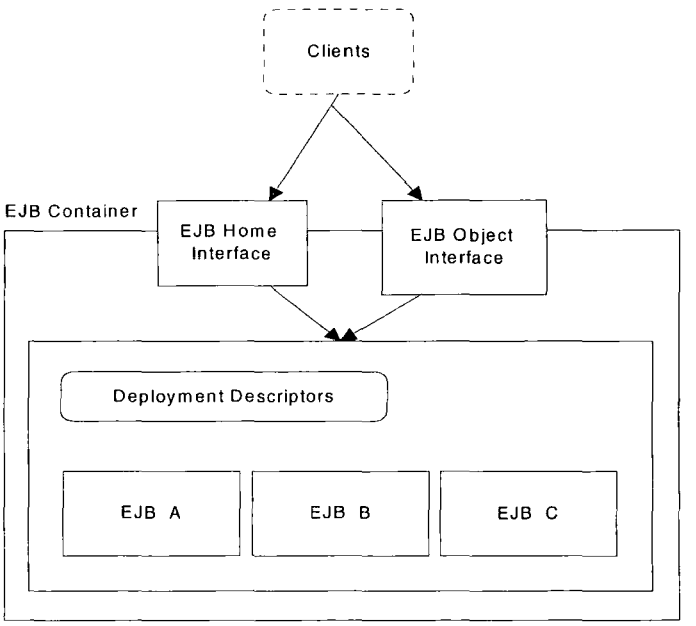


Figure 5 Enterprise JavaBean Model

As shown in Figure 5, an Enterprise JavaBean contains the default features from normal JavaBeans with additional attributes for distributed features and business aspects. Like JavaBeans, EJB are deployed by their containers with separate information for their services; therefore these services can be managed and customised by visual development tools.

In general, EJB architecture focuses on developing three–tier applications, where middleware servers play a vital role. As shown in Figure 6, an EJB architecture can be used with the middleware server to provide services requested by different types of clients. The clients include web browsers and Java applications.

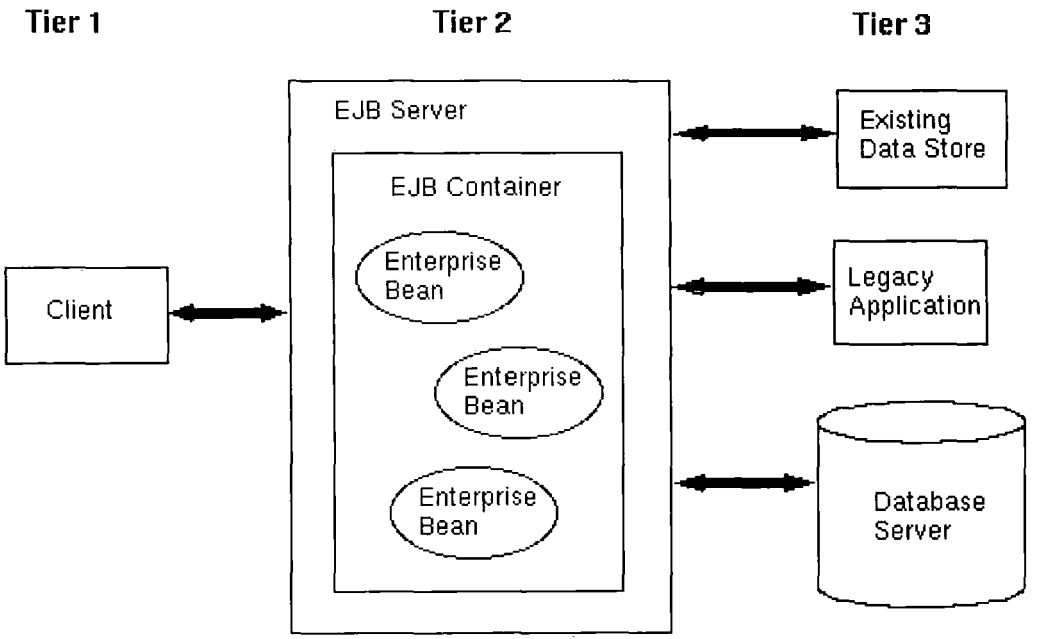


Figure 6 An example EJB Server in three-tier application

2.2.4.3.3 Development steps

The development steps of JavaBeans are varied depending on the different types of JavaBeans. The developer may produce:

- Visual Java Beans for standalone components,
- Enterprise Java Beans for clients,
- Enterprise Java Beans for Multi-tier servers

Therefore there are many different ways to develop Java beans. Since we are only interested in distributed Enterprise beans for clients and Multi-tier server beans, the survey is made on the development steps for constructing EJBs. The basic development steps are:

- Write the Remote Interface Code
- Write the Home Interface Code
- Write the Enterprise Bean Code
- Compile the Source Code needed by the Enterprise Bean

- Create the Deployment Descriptor
- Package the Enterprise Bean

Once the bean is developed, the EJB cannot be directed deployed into an EJB server. Instead the bean has to be imported into enterprise application. An enterprise application may contain one or more of these server-side components: enterprise beans, .jsp files, web files, and servlets. Servets are similar to applets that run server-side to extend the functionality of the server. For the client Beans, the development steps are different. The basic development steps are:

- Locate the Home Interface
- Create an Enterprise Bean Instance
- Invoke a Business Method

Once the client bean is developed, it can be compiled and run on different platforms using appropriate Java compilers to connect with one or more servers.

2.2.4.3.4 Application Domain

There are two different application domains. There are applications which focus on visual programming using JavaBeans and distributed applications that are based on Enterprise Java Beans (EJBs). In general, JavaBean technology can be used to develop JavaBean components, stand-Alone applications, applets, reusable class packages and libraries. An example visual development tool which provides JavaBean is IBM's VisualAge for Java.

Using EJBs, the programmers can interoperate with other technology such as JDBC, RMI, COM, sockets and CORBA. Example applications include Client/Server JavaBeans using JDBC for database oriented applications; two and three-tier applications using RMI and CORBA applications.

2.2.4.3.5 Development tools

Sun Microsystems, which is the main source of the JavaBeans technology, has provided a JavaBeans™ Development Kit (BDK) to support the development of JavaBean components. This tool is more useful for developers who intend to construct application development environments or visual development tools. This tool consists of BeanBox tool, which is a sample bean container that allows a developer to test the functionality of the beans. It also consists of BeanContext container for supporting Bean development. Since this BeanBox is a

sample Bean container which is provided with the source code, the developers can use an example for developing other beans. The latest version of BDK, the Bean builder, can be freely available to download from Sun Microsystems web site.

2.2.5 Other Models

On top of the above four main approaches, CORBA, COM, SOM and JAVABEANS, there are other approaches proposed by different researchers. However they are not as widely used as the above models. At the time of writing, two other models are proposed. They are:

Flexible Object Architecture: This architecture is appropriate for constructing small applications and components. The architecture is based on *flexible components* that are executable objects that can be modified, extended and glued at runtime. This architecture uses component specific programming language that allows components to modify and extend dynamically. It also consists of an implementation of a prototype based on the model called Alego [Leeb, 1996].

Yasmin: Yasmin is also an architecture which consists of a kernel, and software components called droplets. Although the architecture is said to be a general, it is built to create network management applications. It also allows the user to compose software components and add/or replace them at runtime. It consists of two main services: user and kernel. The kernel includes different services which collaborate with each other to provide services for user level. The user level services change from application to application. The user services are implemented using components called droplets. *Liaison*, which is a sample application based on Yasmin architecture, has been developed to validate the architecture [Deri, 1997].

2.3 Component Development Issues

The study of different component models and technologies provide an opportunity to describe common issues surrounding the component development using different technologies.

Many researchers believe that components and interfaces are the leading way to solve many problems with monolithic applications as discussed in [Vigder and Dean, 1996; Szyperski, 1998]. Components allow us to achieve reusability by reusing existing components and to increase modularity. Accordingly the use of distributed components changes the way we build software systems. The components can be custom made (i.e. self made) or ready made off the shelf components. The ultimate aim is to be able to plug in and out binary components as services on the running distributed system with minimum changes. However, there are many barriers to overcome when building applications by integrating the components. The main problems include:

Problems in implementation Languages – At the implementation level, problems with integrating components that are written in different languages is a basic problem that every developer has to deal with and has to overcome by encapsulating implementation details.

Problems in component interfaces – Since components may be designed and developed by independent vendors, there are mismatches between naming conventions, parameters, and control and data flows.

Problems in component communication protocols – Some components are designed and implemented to specific protocols such as C Libraries based on Unix Pipes and DLLs based on remote procedure calls. There is always a need to construct wrappers for interoperability.

Difficulties in component adaptation – Components usually need to be adapted when reusing existing binary components to a new system or adding and modifying their functionality as the system evolves. The component adaptation leads to the implementation of bridges, wrappers and adapters if source code is unavailable.

Problems in Component reuse – Some components require extensive code modification or adaptation to reuse, especially if they are targeted for a specific architecture and communication protocol [Garlan, Allen et al., 1994].

In order to overcome these limitations, many researchers have proposed different ways of component adaptation such as superimposition [Bosch, 1996], adaptors [Küçük, Alpdemir et al., 1998] and subtyping [Hölzle, 1993]. On the other hand, there are a variety of component based distributed models and middleware architectures that have emerged in 90s. Among them, two of the most successful and influential are Microsoft's DCOM and OMG's CORBA.

There are also more traditional approaches such as sockets, TCP pipes and HTTP/CGI approaches for Web based applications. There are also many commercial and research based implementations based on these standards. These standards and associated tools address how to construct and integrate new components based on these standards. But they are lack of providing higher level support for:

- integrating legacy components,
- bridging between different distributed object models,
- adaptation of components in terms of their functionality and interaction,
- mapping between architectural description language (ADL), architecture development environments and tools, and modern distributed models such as CORBA, EJB, and DCOM.

Our main objective is to bridge the gap between many lower level features and services provided by these modern middleware architectures and higher level design support for integration legacy components.

2.3.1 Scenarios

The study of different models has also led to documenting different scenarios where component based development can be applied as a preferred approach over other software engineering methods.

From the applications point of view, the following three scenarios highlight suitable applications that a modern component technology can assist in the development process.

Firstly, component based development may be applied when developing systems that require extensive flexibility. For instance, this scenario may be achieved when designing architectural frameworks for building manufacturing suites for production organisations in engineering sector. The suite may include a set of case tools such as stress analysis tools, etc. and a set of other related tools for assisting the designers. In this scenario, the system can be composed of a set of commercial off-the shelf (COTS) components and other legacy components integrated together to form a component based system. Accordingly, the flexibility of the system is very important since the existing and new components have to be adapted, incrementally updated and added regularly. In this case, features for supporting adaptability and flexibility in the system are more important than facilities that enable dynamic creation of components at run time.

Secondly, systems that are required to add and remove different components at run time to provide different quality of services. These include online banking, process control,

embedded systems and simulation programs, which cannot be restarted or recompiled easily to add or remove components once the system is on-line. In this scenario, the dynamic integration of components is important since there are no point to point based static links between components. Accordingly such systems require flexible plugs, connectors, or adapters as interfaces, together with a configuration manager, which allow modification and extensibility of the components within the running system.

Thirdly, systems that require monitoring of their components such as e-commerce systems and web based distributed systems. These systems are highly distributed and consist of a large number of users. The monitoring includes different services such as load distribution, measuring performance, fault tolerance, and security of components. Therefore individual components need to be wrapped for adaptation and monitoring of components. In this scenario, different monitoring services should also be able to be dynamically added or removed without modifying the functionality of the component.

Some systems may require all of the above scenarios. The three scenarios highlight different factors that influence the component based development. To summarise, some component based systems require adaptability, and flexibility and others might need extensibility and services as the main concerns in the development.

2.3.2 Component modeling for composition

The software development using component based approach may be traditional This includes analysis, design, implementation and assembly, and deployment. During the implementation stage, it is different from building Object Oriented systems because systems are built by component assembly rather than new development. Therefore all the problems with designing and implementing different phases of the development have shifted to composition and assembly of components to form the system. In other words, during the implementation, developers are spending more time composing components rather than developing them. When composing existing components, interfaces of the components needed to match *syntactically* as well as *semantically*. Hence, in the literature, various component composition methods have been proposed that focus on matching specifications of the component interfaces for interoperability amongst existing components. This includes superimposition [Bosch, 1996], wrapping, component interface adaptation, filters, and semantic interfaces [Aksit, 1996] [Penix and Alexander, 1997].

Furthermore, the matching of the requirement specification and the specification of a component interface determines the possible reusability of the component. Accordingly, there are various research proposals that describe matching methods, as presented in [Zaremski and

Wing, 1997]. As the need for describing component composition within a system is recognised, various formal and semi-formal component languages have also emerged [Achermann and Nierstrasz, 2001].

The component composition methods addressed above by various researchers depict implementation level component composition. This research does not intend to introduce a new component composition model or language. However the AbCD approach intends to facilitate the modeling of components for easier component composition, thus the components are reusable. It is achieved by allowing developers to define context-based aspects and abstraction levels of the elements of the interface together with component functionalities, when defining component interfaces. This adds the meta-data for component interfaces, thus allowing automation of component composition. Adding meta-data to component interfaces may allow components to be semantically modeled for transformation into target platform specific components. This also means that tools can be used to automate the adaptation or transformation process.

2.4 Defining a common framework for components

From the literature it is possible to categorise different types of components as shown in Figure 7. This is a summary of the detailed description of different types of components:-

- Generic components: They are referred to as monolithic components that do not rely on application component frameworks or have their own custom frameworks. They can be application specific components or generic library components such as a Web server, a Database or a XML parser. They generally provide functionality using APIs, or may have an independent execution or runtime environment. In other words they can be shared components (i.e. standalone) or private (i.e. library) components. Although these components are generic, the interoperability of these components is limited by the operating system that they support and the programming language that they are developed in.

- Desktop components: The concept of component based development was first realised with the form of desktop components such as JavaBeans. The Java programming language has integrated support for the JavaBean model. Based on OCX and COM technology, there are many desktop components to work with Microsoft Windows applications, mainly in the form of visual components such as toolbars or forms. There are also utility components such as File processor or report generator components. There are also other early component frameworks such as the BlackBox component framework and IBM's System Object Model (SOM)

frameworks that support the development of component-document based components. These components can also be regarded as desktop components as they focus on rich GUI applications Desktop components are generally designed to be fine-grained components and they can be implemented as inter-operation within the same process, that is in a single desktop application, rather than distributed applications).

- Business components: Components that are built using modern distributed technologies such as J2EE, COM+, and Web Services. Each technology has component frameworks to allow developers to create business components. Szyperski has described the software frameworks: "A component framework is a software entity that supports components conforming to certain standards and allows instances of these components to be 'plugged' into the component framework..."

  The three frameworks described in this literature are referred to as heavy weight frameworks as they applied an all-in-one approach of providing component services. On the other hand, business components can be built using light weight frameworks such as the Spring application framework and the PicoContainer framework [Harrop and Machacek, 2005]. These technologies provide non-invasive frameworks such as Inversion of Control (IoC) container. In the next two sections, the detailed description is made on how building components on light weight frameworks is different from heavy weight components.

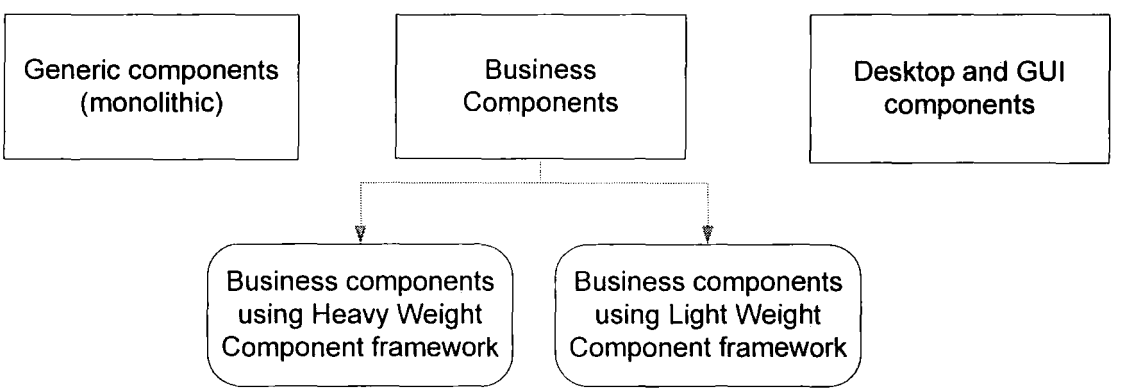


Figure 7 Types of components

This research focuses on the modelling of business components as specification components at an abstract level. Accordingly the meta-model described in the next section is to support the modelling of components that are based on different type of frameworks. However, before describing the meta-model, it is necessary to address the relationship between the target

component models that will be designed on the meta-model and different component frameworks. From the modelling point of view, OMG's proposal for the transformation of PIM to PSM can be further refined into PSM for Heavy Weight or Light Weight component frameworks, as shown in Figure 8.

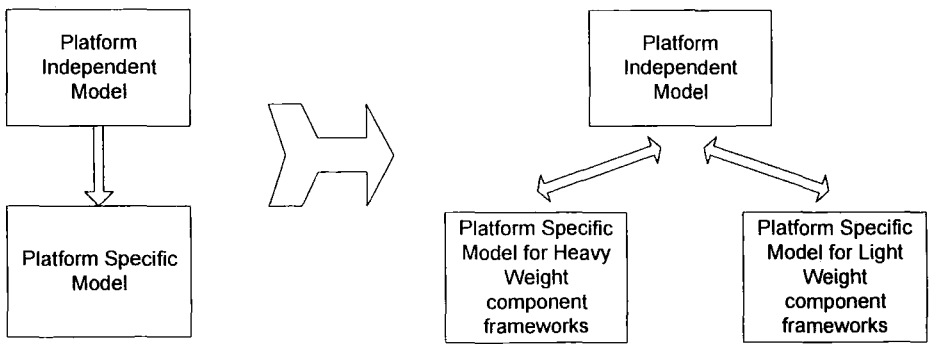


Figure 8 OMG's Proposal for PIM and PSM (left), Component modelling for two different types of component frameworks (Right)

From this figure, it is important to note that the different type of component frameworks can dictate the way PIM to PSM is transformed. Further analysis on the use of Heavy weight and Light weight frameworks is discussed in the following sections.

2.4.1 Component construction for Heavy Weight Frameworks

Three main organisations, who are major players of component technologies, OMG, Sun and Microsoft, have adopted different standards (also known as 'wiring' standards). Each standard has its own framework to support the standard. These frameworks provide a variety of ***component features*** such as remoting, lifecycle management and ***component services*** such as transaction management, and security. These component features allow business components to achieve abstraction such as location transparency, and language transparency. However these frameworks can be regarded as invasive and heavy weight. This is because when using the framework, the design and architecture of the component is dictated by the wiring standard and the implementation of the component relies on the runtime environment provided the framework. As a consequence, the business functionality of the component is embedded within the code that provides framework dependent functionality. In [Szyperski, 1998], Szyperski has referred to these three heavy weight frameworks as "The OMG way", "The Sun way", and the "The Microsoft way". Although these standards have their own implementation frameworks, it is possible to describe a set of common concepts that all frameworks support for component development. Although there main heavy weight technologies, i.e. .NET/COM+, CORBA/CCM and J2EE, have been presented in the literature survey, Table 1 summarises the common concepts that are shared amongst them.

	Concepts	.NET/COM+	J2EE	CORBA/CCM
Design and architecture	Classification of components	N	Y	Y (with CCM)
	The use of Containers	Y (with COM+)	Y	Y (With CIF)
	Composition with Contexts	Y (with .NET enterprise services)	Y	Y
	Indirection	Y (with context proxies)	Y (with remote/local proxies)	Y (with stub)
	Interface Definition Language	Y	N	Y
	Non-invasive approach	N	N	N
	Attribute (declarative programming)	Y (with .NET framework)	Y (With new Java 1.5)	N
	Component	N	N	N

	composition using IoC			
Component Management	Container features	Y (with COM+)	Y	Y (depending on CCM implementation provider)
	Component Services	Y (with	Y	Y (depending on CCM implementation provider)
Deployment and Runtime	Packing and assembly	Y (including version control and strong naming)	N	Y (with CCM)
	Distributed environment	Y (with .net remoting)	Y (with RMI)	Y (with IIOP)

Table 1 Common concepts in J2EE, .NET/COM+ and CORBA/CCM

This table highlights the fact that heavy weight frameworks focus on providing component management features such as container, context management and component services. However, they lack the necessary support for component design such as component composition and dependency.

2.4.2 Component construction for Light Weight frameworks

Using light weight frameworks, implementation of a component is regarded as a plug-in and is completely separate from the interface. Like other component based frameworks, a light weight framework includes a container that manages instantiation of objects and dependency between them. It is known as an Inversion of Control (IoC) container. The IoC container uses the configuration settings to identify which plug-in, i.e. its implementation, to reference and use at runtime. In other words, implementation dependency is formed dynamically at runtime.

This is because light weight frameworks are based on non-invasive approaches. This is different from constructing components using Heavy Weight frameworks. In this way, a component designer can construct components, for example using POJO rather than EJB components, and still take advantage of component services such as persistence and transaction processing. When applying light weight frameworks, container features such as context, and interception using proxies are not part of the container and can be applied as modules. Furthermore, components services such as persistence and transaction management are also modular, and not part of the framework. Therefore it is the developer's responsibility to apply different component services from other 3rd party services and apply dependency injection on the implementation. Light weight framework based technologies like the Spring

Application Framework [Harrop and Machacek, 2005], and the Castle Project [Avalon, 2005], provide different component services as pluggable modules.

Accordingly, from a component design point of view, transforming PIM to PSM is different if one is planning to use a Light weight framework for implementation of the component. This is because there are fewer dependency relationships between application components and framework components that provide component services. Using a light weight framework, the service dependency relationship between components is indirect. This is achieved by registering different framework components and application components with the container and explicitly defining the dependency using a set of configuration settings.

To summarise, light weight frameworks add two essential values to component based development:-

- It encourages the developers to use interface dependency rather than implementation dependency by providing an IoC container that uses the dependency injection pattern.
- The reusability of the components is improved because it focuses on providing non-invasive design.

2.4.3 Summary of the software components and frameworks

To summarise, component development concentrates on the writing of many existing and new components using different frameworks and standards, whereas traditional software engineering focuses on the production of monolithic systems. In other words, the construction of a component based system includes wiring of components using a component framework. The frameworks are based on different wiring standards. These standards are derived from different background areas of software development and are designed for different types of applications. Therefore, when designing a component, the designer is faced with not only domain and application aspects but also with the constraints surrounding the components standards for interoperability. The component design can then be targeted to use light weight or heavy weight frameworks. Table 2 summaries the main differences.

Light Weight Frameworks	*Heavy Weight Frameworks*
Modular component services	All-in-one component service
Based on the principle of non-invasive approach using IoC and component services as plug-in modules	Based on the principle of providing component framework features such as lifecycle management, pooling, persistence, etc.
Supports component composition through dependency injection using IoC container and composition configuration settings.	Supports service locators provided by component framework for integration of components.
Application developer has to import different 3rd party modules to provide component contexts.	Framework support for component contexts.
The application component does not have to comply with container API.	The application component must follow the container API guidelines.

Table 2 Different focus areas of Light Weight and Heavy Weight Frameworks

From the description above, it is possible to form a common pattern from different component frameworks that support the construction of business components. Figure 9 shows a simple pattern with a high abstraction level. All component frameworks provide a form of contextual component composition. A ***context*** can be at application level or at a session level. Context based composition helps the container to provide necessary runtime environment for its

components. In other words, components that require a particular constraint such as interception or transaction management can be grouped into a particular context for processing. Some technologies provide context support as part of the programming language, such as Common Language Runtime (CLR) from Microsoft. However other technologies, such as J2EE, use deployment descriptors to configure the container to set a particular context.

A container also provides a factory for components using a form of ***factory proxy***, although implementation can be varied depending on the framework implementation. A framework also provides a form of component proxies as an indirection of access to the services of the component instances, i.e. ***instance proxy***. For example, frameworks based on J2EE standards provide remote as well as local proxies for allowing remote and local clients to access the service. However light weight frameworks, such as Spring, only provide a proxy as an additional plug-in for interception and AOP services via configuration.

Both heavy weight and light weight component frameworks provide various ***component services***, such as transaction management, security, logging, caching, and persistence. Depending on the framework, it can be as integrated with framework implementation or a third party plug-in via configuration. Heavy weight frameworks, such as CORBA/CCM and J2EE, provide services integrated with the framework. However, light weight frameworks apply them as plug-in implementations.

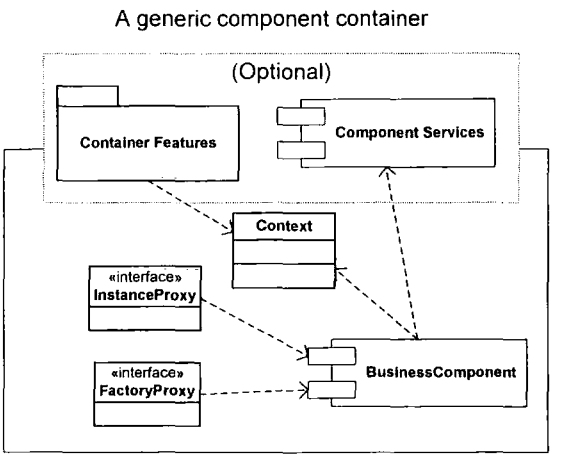


Figure 9 A sample common component framework

Similarly, current technologies that implement component frameworks provide various ***container features***, such as event propagation facility, remoting facility, interception facility, and component factory facility. Using a heavy weight framework, these features provide integrated within the infrastructure of the framework, whereas light weight frameworks use a modular approach.

Chapter 3 Model Driven Development

3.1 Introduction

This chapter describes a different view of the software development, which is the Model Driven Development (MDD). MDD is studied in this research because it provides a way to specify components in abstract and logical way, and possibly encapsulate component technology details. This chapter presents MDD approach proposed by the Object Management Group (OMG) [OMG, 1998]. Further more it discusses how UML meta-model can be extending to support different platform specific domains. This chapter also also describes the Aspect-Oriented Programming (AOP) concepts and how it can be applied in component development. It provides an example as a simple case study to illustrate the concepts.

3.2 Model Driven Development (MDD)

In 2001, Model Driven Architecture (MDA) was introduced by OMG [OMG, 1998]. This section describes MDA and also introduces other model driven development approaches in the literature. Before continuing to address detailed principles, it is necessary to define terms and definitions of the model driven development that will be used throughout the thesis.

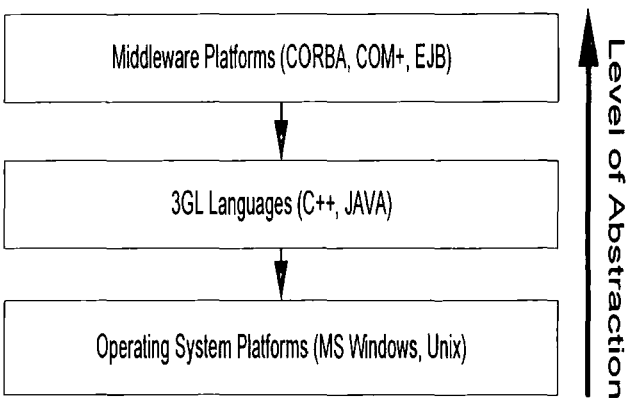


Figure 10 Middleware technologies and 3GLs

As shown in Figure 10, with the emergence of *Middleware* technologies, *level of abstraction* has increased from writing *system platform*[1] specific implementations with 3rd Generation Languages (3GLs) to system platform and programming language independent implementations [Pritchard, 1999]. Table 3 summaries the most well known and accepted middleware technologies currently in the literature:-

Company	Technology
SUN	Java 2 Enterprise Edition (J2EE) based on Enterprise Java Beans (EJB)
OMG	Common Object Request Broker Architecture (CORBA), CORBA Component Model (CCM)
Microsoft	Component Object Model (COM), COM+, .NET

Table 3 Summary of Middleware technologies

Often current middleware technologies also support the concepts and principles of Component development as described in the previous section. Although Middleware technologies have increased the level of abstraction, each technology proposes its own *platform*[2] specific standards, runtime environment and infrastructure. Accordingly, developers have to follow the newest technology's specific standards to take advantage of its provided services and facilities. However, as the technologies evolve overtime, developers are forced to adapt their systems to new standards for *portability*. Furthermore, bridging or porting is required amongst different technologies for *interoperability* and to resolve *architecture mismatch* [Garlan, Allen et al., 1994]. The CORBA middleware technology is a prime example of such case. As CORBA was introduced as a specification, different vendors constructed implementation frameworks based on the specification. However components implemented with one implementation framework were not able to communicate with others due to object referencing incompatibility. Furthermore, the *reusability* is reduced because technology specific code is embedded within the code that performs business processes. To overcome this changing and evolution of technologies and to separate the business process code from technology specific code, a higher level of abstraction is required that is technology independent. One possible solution is to construct platform independent models as *software artefacts* and convert them into platform specific codes as needed, using automated code generators. MDD is based on concept of producing such models as software artefacts.

Before describing the components of MDA in detail, it is necessary to discuss the role of models in software engineering lifecycle. Based on the work presented by Daniels and Brown [Kleppe, Warmer et al., 2003; Brown, 2004], Table 4 presents an evaluation of how modelling approaches are applied in various software development projects.

[1] System platform is referred to as operating system platform and its associated environment.
[2] Platform is referred to technology specific infrastructure such as middleware technologies (i.e. COM, CORBA, etc.)

Usage	Description
Code only	Modelling is formally not used in the development. In most small to medium projects, developers believe that modelling is an unnecessary extra step in the development. Modelling languages, such as the UML, may only be used to clarify the understanding of a particular problem on paper, or in documentation. It may also be used to present an overview of the system architecture.
Model » Code	Modelling is used to design the system. In other words, models are treated as *first class* artefacts and a model driven approach is formally used. In this case, generic modelling languages, such as the UML, provide facilities to model system requirements, static structure and dynamic aspects of the system. This is the area where software development projects may take the benefits of the MDA, where various code generation tools are used to automate the process of transforming platform independent models to platform specific code. [www.codegeneration.net]
Model » Code » Model	In some projects, models are used to present the business requirements and overview of the system design. The models are then converted automatically using tools, or manually by the developer to platform specific implementation. It is then converted to detailed design models that include platform specific representation of the models. This is also known as *round trip engineering*. Commercial companies such as IBM and Sun provide a set of tools to support round trip engineering. [Rational Rose, Java studio]
Code » Model	Code visualisation plays an important part in program comprehension. Code can be *reverse engineered* to models to provide more understanding of the static structure and dependency of software modules. Most commercial and open source tools provide reverse engineering facilities as well as simultaneous views for model and code. [JBuilder, Visual Studio]
Model only	Models may be used only to present business processes, business requirements, design patterns, system architecture, and business enterprise models.
Model » Model for CbSE	In addition the usage of models described above, this research focuses on transforming platform independent models that represent business concepts to models that represent components based on the principles of CbSE. In other words, components are modelling artefacts rather than implementation code within the software development.

Table 4 The use of Models in Software Development (Based on the diagram presented in [Brown, 1996])

The following section describes the principles and properties of MDA and discusses how it can be extended to fit CbSE.

3.2.1 OMG's MDA

Model Driven Architecture (MDA) is based on the principles of constructing models for the development software using well defined notations. As described in the previous section, it is very important to derive platform and technology independent models because models are used as an abstraction to technologies and platforms. OMG's MDA achieved this by defining

meta-models, *models*, *modelling notations* and *model transformation rules* [OMG, 1998]. Before continuing the detailed discussion on the principles of MDA it is important to clarify the definitions of these terms in the context of this thesis. In MDA, it is defined as :-

> "a model of a system is a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text. The text may be in a modelling language or in a natural language."

Others state that "A model is a description of a system written in a well-defined language" [Kleppe, Warmer et al., 2003]. A more generic definition was found as "a simple and familiar structure or mechanism that can be used to interpret some part of reality" [Boman, 2004] and "Models are used to reason about a problem and design a problem domain and design a solution in the solution domain." [Brown, 2000].

From the above definitions, it is noted as the definitions are relative and at different abstraction levels, as addressed by [Ivan, 2001]. In the context this literature, the principles of modelling in software engineering are presented as follows rather than defining the term 'model'.

- A model can be regarded as a representation of the system under study. This *context of the system* in this case is relative. It can be a problem case, a component within the system, or a particular view of the system.
- The model is constructed using a well defined *modelling language*. The modelling language may use formal and/or informal language.
- The model may have different *abstraction level*. The model may have different *views* on the system.
- The model may focus on different *aspects* of the system. This has broader meaning to the model in compare with what was defined in OMG's MDA, as will be described in later sections.

A model can have different roles in relation to the system that is being represented, as described previously. When a common problem case or a design pattern is modelled as general use and reuse, it can be regarded as "systems as models". However, modelling a particular system for understanding may be regarded as "modelling the systems". This research follows the former role as models are made to represent software components for greater reusability.

Figure 11 shows an overview of the MDA approach proposed by the OMG. MDA defines meta-model which is a model used to construct modelling languages. MDA also offers the Meta Object Facility (MOF), which is used to define meta-models. One of the most successful meta-models is the Unified Modelling Language (UML), which can be used to construct models of the system [Pooley and Stevens, 1999].

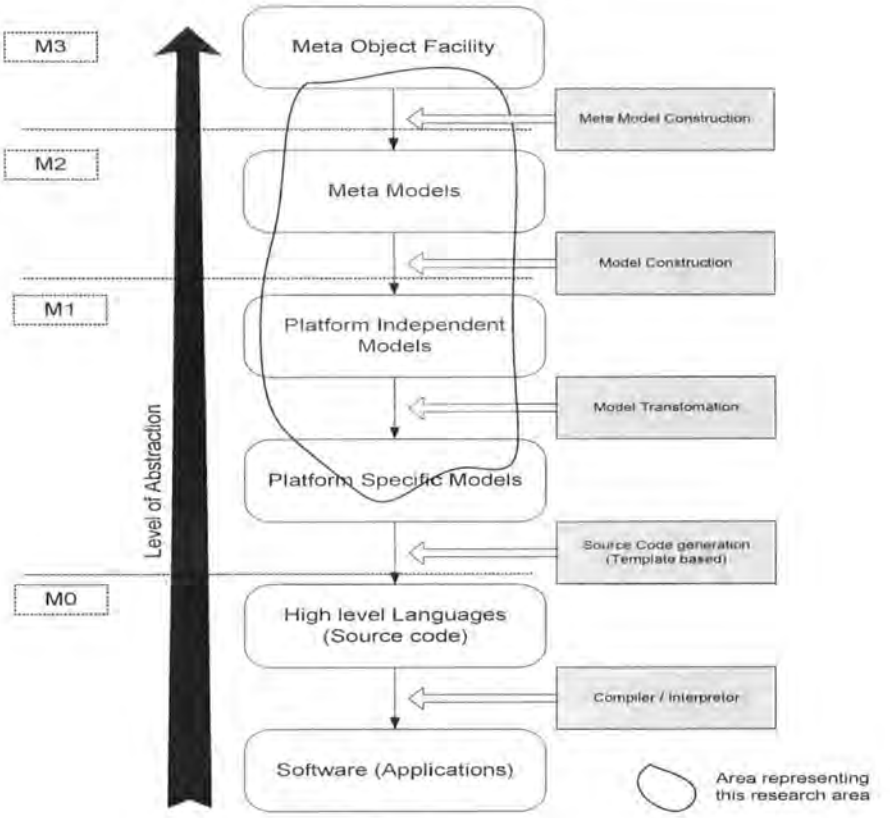


Figure 11 Model Driven Architecture and the level of abstraction

MDA also defines four-layer architecture. Starting with Meta-level 3 (M3) where MOF is used to construct meta-models such as the UML. Meta-level 2 (M2) contains meta-models defined using MOF. In other words, Meta-models are *InstanceOf* MOF constructs. These meta-models can be generic such as the UML or can be domain specific meta-model such as Common Warehouse Meta-model (CWM) for data modelling, or Software Process Engineering Model (SPEM) for process modelling [(OMG), 1999]. Meta-level 1 (M1) models are instance of meta-models such as UML class diagrams for particular application. Meta-level 0 (M0) is generated or implemented instance of M1 level. The detailed of each meta-level and the focus of this research in each level is described in the following sections.

3.2.2 Meta-modelling and Meta Object Facility (MOF)

MDA defines a meta-model as "model of models". In the world of MDA, a model is referred to as "InstanceOf" of a meta-model. For example, a UML model for Doctors' surgery system is an InstanceOf a UML meta-model. A UML model itself is referred to as the "ModelOf" Doctors' surgery. MDA provides MOF, which is a meta-modelling language to construct meta-models. While MDA only provides the meta-modelling language and others provide a wider view of meta-modelling to include *processes*, as described by [Gigch 1991][Brinkkemper, 2000].

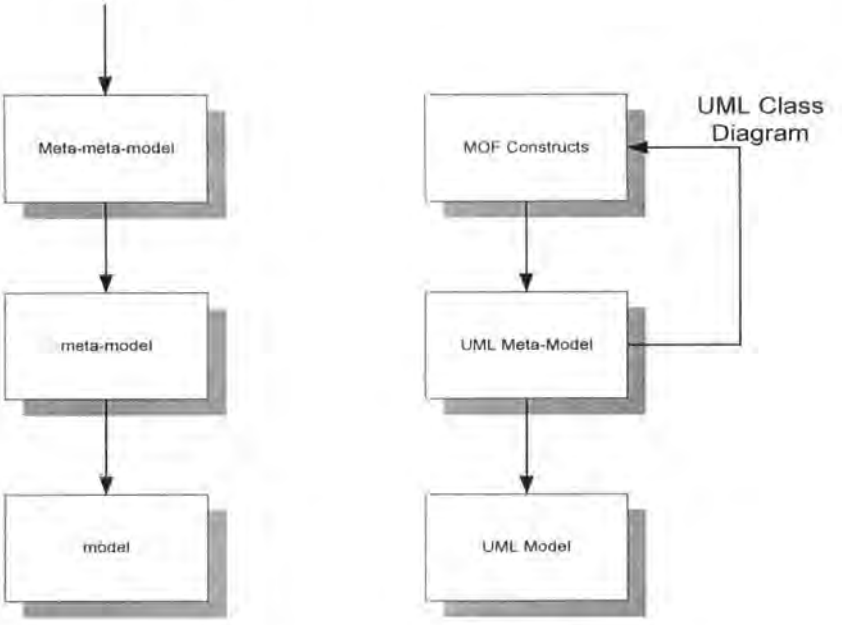


Figure 12 Meta-modelling levels (left) and OMG's MDA approach for meta-modelling (right)

Figure 12 (left) shows an example meta-modelling approach. A meta-meta-model can be constructed using meta-meta-meta modelling constructs, a meta-model can be constructed using meta-meta modelling constructs, and so on. However to avoid having to introduce new meta languages and their syntaxes as the hierarchy goes to more higher levels, MDA uses a subset of UML class modelling constructs and state chart modelling constructs from the UML meta-model, as shown in Figure 12 (right). This is used as the MOF abstract syntax for meta-modelling. This means that UML modelling tools can be used to construct meta-models such as UML meta-model and other meta-models for different domains. In other words, MOF is self-reflective and all the levels above MOF may be treaded as MOF, i.e. in a uniform way with reflective APIs. Using the MOF as a standard, OMG proposes meta-models for other

domains such as Common Warehouse Meta-model (CWM) for data warehousing domains, and CORBA Component model (CCM) for CCM technology specific models.

MDA also defines the notion of *abstract syntax* and *concrete syntax*. Abstract syntax represents the concept of the model and its elements. The abstract syntax can be viewed or presented using concrete syntax such as UML graphical notations or XML Model Interchange (XMI) language [OMG, 1999]. In other words, a designer can create abstract syntax for a new meta-model using a UML tool that supports UML notations and XMI functionalities. This separation promotes model transformation, interoperability of MOF compliant meta-models across domains and integration of tools, which will be discussed later in the chapter.

3.2.2.1 UML Meta-model and MOF

Figure 13 shows the UML Class model element and its relation to other model elements as an abstract syntax. As described in previous section, it uses UML class modelling constructs. MOF offers the following five modelling constructs to define a meta-model.

- Model Types (i.e. class, data types, and enumeration)
- Attribute (i.e. Class properties)
- Association (i.e. aggregation, composition)
- Generalisation
- Operations

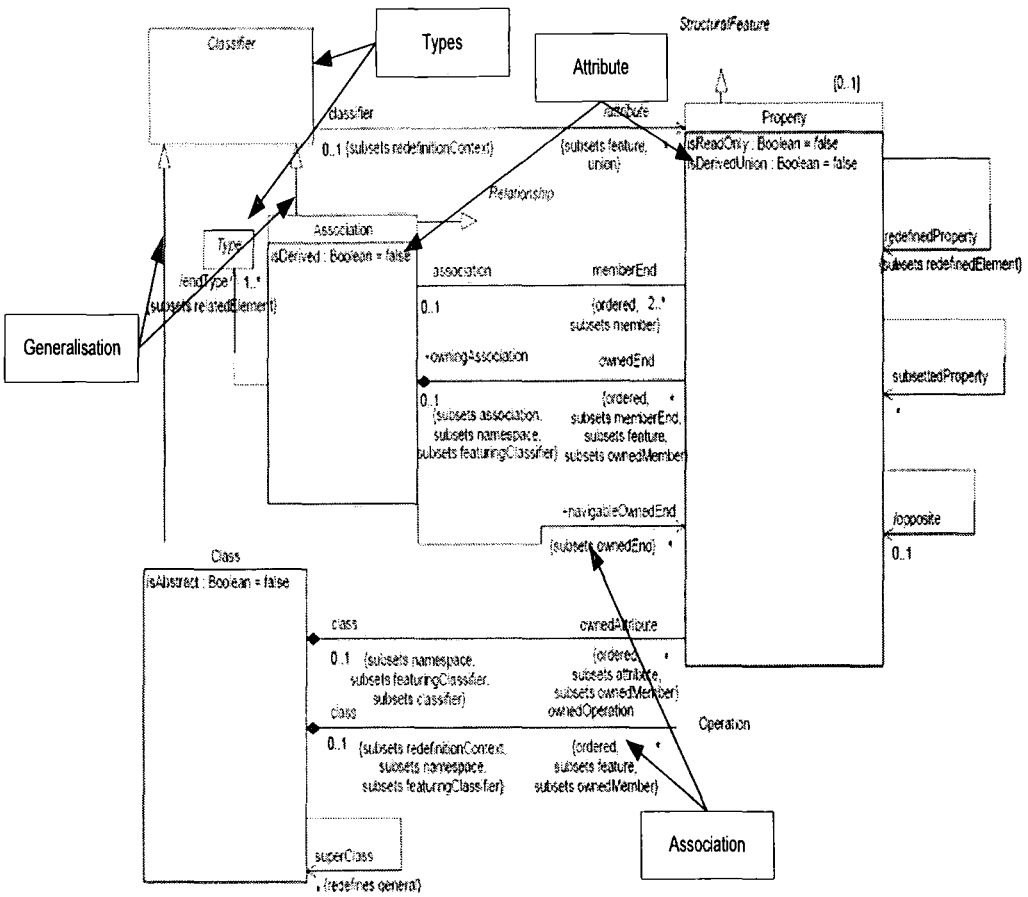


Figure 13 A fragment of UML meta-model (From OMG's UML Infrastructure Meta-model)

The use of UML class modelling constructs in MOF specification makes the MOF meta-models more transparent. With the use of UML modelling tools, better tool support for creation, transformation and automation of meta-models can be achieved. Currently OMG provides three types of MOF mapping.

They are :-

- MOF Mappings using XMI
- MOF Mappings using CORBA IDL
- MOF Mappings using JAVA JMI

3.3 Attribute and Aspect concepts in Software Engineering

Adding notes or labels to an object to indicate what has been done or what has to be done with that object is not new in the real word. Adding notes or attributes to a software component, however, is new in software development paradigm. Generally attributes are added to program elements at implementation stage to indicate that meta-data is added to that element. Program elements can be any artefact that is part of the program code, such as a class, a method or a property. This is also referred to as 'Attribute-based programming' or 'Attribute-oriented programming'. The metadata or information added to the program element can be domain/application specific, technology specific, or system specific. Some of the common attributes that can be added to a program element are listed in Table 5.

Example Attributes	Domain/ application Specific	Technology/ platform Specific	Development Specific
Logging			*
Remoting		*	
Relation	*		
Persistence	*	*	
Security	*	*	
Activation	*	*	
Transaction	*	*	
Clustering	*	*	
Excepting handling			*

Table 5 Non-functional requirements

As listed in Table 5, software developers need to put together different **aspects** of the software to form a working system. These aspects are also referred to as 'non-functional' aspects of the system. The main problem is, as these aspects re-occur in many different applications, developers need to re-write the same or similar pieces of code again and again

with different technologies or programming languages depending on the application requirements. Therefore they are orthogonal to the specific application.

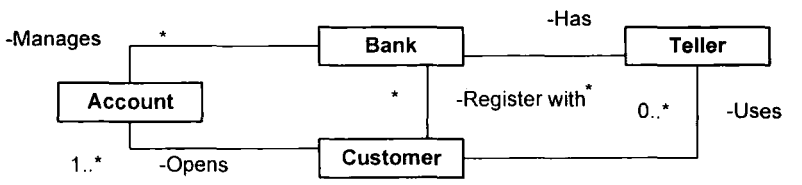


Figure 14 Simple Case study

As show in Figure 14, a simple banking example can be used to illustrate this problem. Different Use Cases are used to identify common aspects of the development. For instance:-

- Use Case 1 : Bank manages accounts : Transaction and Database processing is needed to process accounts.
- Use Case 2: Bank manages customer details : Security is needed to authenticate customers.
- Use Case 3 : A Customer must deposit minimum of £1 to open an account.
- Use Case 4 : Customer uses Teller machines : Secure channel is required to process transactions.
- Use Case 5 : Any changes to account and customer must be recorded for historical purposes.

In the above use cases, UC 1, UC 2 and UC 4 can be regarded as non-functional aspects of the system. The same aspects of concerns apply to many different application domains. There are also some aspects that cannot be encapsulated with a single class. These aspects are concerns that may exist across different places throughout the hierarchy of classes. In the above example, UC 5 aspect has logging and tracing concerns that cannot be easily modularised. Accordingly they are referred to as 'cross-cutting concerns' [Laddad, 2003]. This mixture of concerns leads to **redundant** and **scattered** code, which leaves the code for different concerns scattered across multiple classes. As the result, the code becomes:-

- difficult to maintain,
- difficult to reuse,

- and unclear to see the structure.

When one has to deal with many different aspects of the system, one often finds that the 'separation of concerns' is difficult to deal with. In 1968, Dijkstra discusses about the separation of concerns as follows:-

".. one is willing and able to study in depth as aspect of one's subject matter in isolation, for the sake of its own consistency, all the time knowing that one is occupying oneself with one of the aspects." [Dijkstra, 1968]

Traditional Object Oriented Concepts cannot deal with cross-cutting concerns because modularisation is achieved by encapsulating concerns with a class, a package or a component. Sometimes a class may have code fragments that have nothing to do with its functional aspects, but rather to do with non-functional aspects such as logging, tracing, distribution, etc., thus lost its encapsulation and modularity.

3.3.1 Ways of handle cross-cutting concerns and non-functional concerns

Although there is no silver bullet to resolve cross-cutting concerns of different aspects, in the literature, there are many different ways to resolve as general solutions. Gamma has described the used of design patterns such as 'the Visitor' and 'the Observer' patterns to separate concerns [Gamma, Helm et al., 1995]. As the use of design patterns to resolve the cross-cutting concerns is achievable it is not generic and efficient to handle the concerns.

3.3.1.1 Aspect-Oriented Software Development (AOSD)

Aspect oriented software development is focused on adding aspects on top of traditional object-oriented software development. It is not intended to replace object-oriented software development, but to complement by adding a new dimension for cross-cutting concerns.

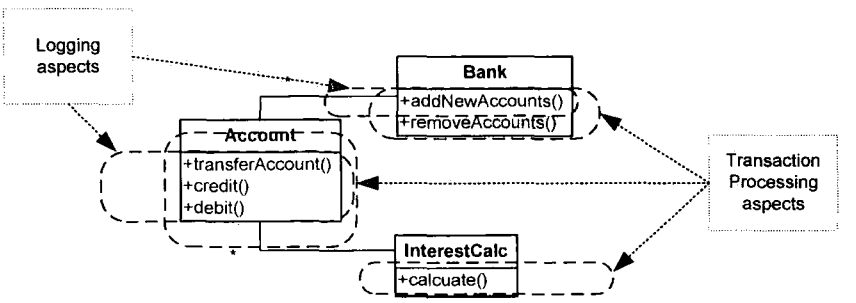


Figure 15 Functional and Non-functional Aspects

As you can see in Figure 15, a system is composed of classes that contain groups of methods that address concerns for **functional** aspects, such as managing accounts, adding accounts and calculating internets. These functional aspects can be referred to as vertical aspects or vertical dimensions. The class hierarchy that is based on object orientation only encapsulates functional aspects within classes and other dimensions for horizontal aspects or **non-functional** aspects such as logging, and transaction process might be spread over different classes in the system. The researcher and developers have addressed these issues by forming a new way of modularising the objects into aspects, i.e. Aspect Oriented Programming (AOP) [Kiczales, 1997]. The concepts derived from AOP can be used to explicitly describe components that require or provide non-functional services. In this way, a component can be described in a more abstract and self-contained manner.

When identifying non-functional aspects, it is possible to classify them into two different contexts. Firstly, there are non-functional aspects, such as logging, persistence, and security, that can be applied by using services provided by component frameworks. Identifying these aspects as logical components in the early state of the design may help the design more independent. Furthermore, it is also possible to describe how the implementation frameworks provided by different technologies can support to these non-functional services. Chapter 4 describes an approach to explicitly define such non-functional aspects.

Secondly, there are also non-functional aspects, such as performance, availability and reliability, can be identify as explicitly as logical components in the design. However as they are not explicitly supported by component frameworks, it is developer's task to define how such non-functional aspects are addressed in the design as they are not supported by current component frameworks.

3.4 Summary of the current literature survey and Model Driven Development

To summarise the background of the research, one area that received more attention is the relation between the development of applications that are domain specific and standard component-based frameworks and their supporting technologies, such as .Net, J2EE and CORBA/CCM. This area is important for several reasons. When we design a modern component-based system, the architecture of the system is formed as a result of the composition of components. These components can then be implemented by applying a target modern component framework. The technologies, that are the implementation of these frameworks, provides a bundle of facilities and features that are domain independent but

enterprise system tailored, such as security services, transaction services, interception and monitoring services.

However, one of the tradeoffs of applying such technology is the compilation of their business models and their implementation model in to the framework. In other words, the developers have to follow a specific component framework (such as component interaction model for J2EE technology, component lifecycle for J2EE for technology, etc.) and its implementation technology during the early stage of the development to design business models and to take advantages of their services. The detailed descriptions of different component-based frameworks are already presented earlier in the Chapter.

OMG's Model Driven Architecture (MDA) has emerged, as a non-proprietary technology, to provide a middle way and bridge purely business models to technology specific models. As a part of MDA, Meta Object Facility (MOF), which is a meta-meta-model, was introduced that allows developers to construct platform independent meta-models such as UML meta-model and other platform specific meta-models.

These meta-models can be used to develop models of the software. A simple example is illustrated in Figure 16. In this example, an instance of platform independent UML business model for a particular application is constructed using the UML meta-model. This model can then be transformed into technology and platform specific component models.

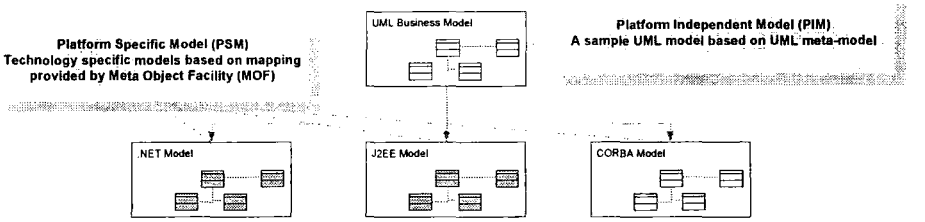


Figure 16 A sample UML and MOF mapping

The emergence of MOF allows MOF compliant tools to automate the process of generating platform specific code based on model mappings.

Model driven development approaches, such as Model Driven Architecture (MDA), only focuses on modelling functional aspects of the system, and lacks modelling of cross-cutting aspects. On the other hand, most of the research focus areas of AOP have been on applying various methods to the code at the implementation level [Laddad, 2003]. In Section 3.3, the literature survey highlighted the need for incorporating AOP concepts to modelling to allow

the development of more reusable component models, at the requirement analysis and design level. Figure 17 shows a common component development model where horizontal services or cross-cutting services are addressed at the implementation level with technology specific frameworks.

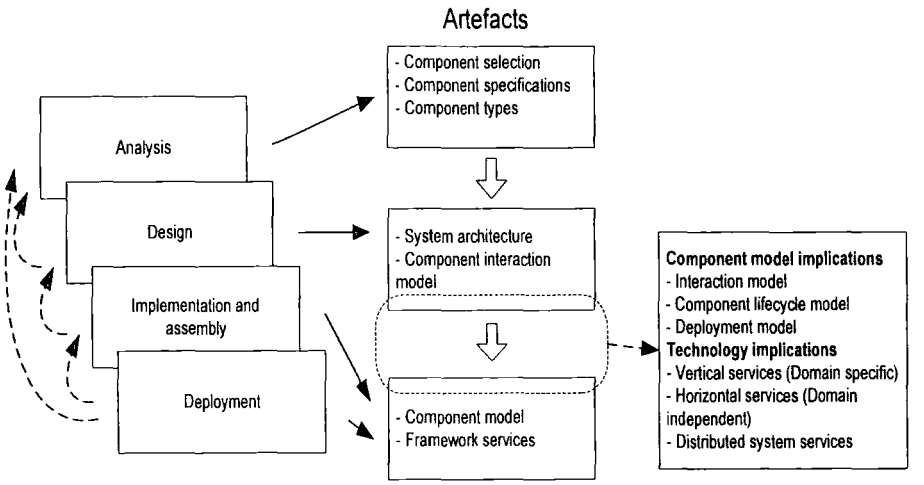


Figure 17 A common development processes using a component based framework

As the figure depicts, a common development process using a component-based approach involves the construction of different artefacts at various stages of the development. The figure also shows that, during the development, component artefacts that are produced or reused have to be based on a component model and its related framework. The component models provide the architectural details such as interaction model, lifecycle model and deployment model, as well as distributed system services. There are also **component model implications** as well as **technology implications** with choosing an implementation platform. Accordingly, when developing component-based systems, the construction of specifications and the design of components at the design stage require detailed knowledge of the component framework and its supporting technology to be applied at the implemented stage. This is one of the open problems of modern component-based system development. Using the model driven approach, this research fills this gap by introducing a new approach, called Attribute-based Component Design (AbCD) approach. This allows the component developers to construct specification-based component artefacts as logical model components that are component platform neutral, yet providing attribute-based model constructs to be able to implement using a targeted component platform dependent technology. The AbCD approach also uses the concepts from AOP to provide facilities for specifying common behaviours of the logical components or "cross-cutting" concerns.

Chapter 4 Attribute based Component Design (AbCD)

4.1 Introduction

The main objective and research contribution here is to support the modelling of software components in the context of CbSE. In this chapter, the research areas presented in the literature survey are summarised and the requirements for a new framework to support modelling of components are elaborated. To meet the requirements of the thesis identified earlier, this chapter then presents a description of how a new model driven approach, referred to as Attribute based Component Design (AbCD) approach, is derived from existing approaches in the current literature. It also includes a simple example that illustrates how the approach is applied.

The principal end products of this research are also discussed in the light of meeting these requirements presented. It also presents a generic view of the different modern component technologies to summarise the details described in the previous chapter.

4.1.1 Background and Aims

Many organisations are trying to implement or update their systems in such a way that such these systems or subsystems can be updated incrementally to keep abreast with new technologies and to take advantage of them. These organisations demand not only sound architectures but also efficient ways to reuse existing in-house as well as third party components.

In the field of component-based development environment and enterprise computing, many researchers are focusing on developing new component-based co-ordination models with their own component types and integration methods [Alder, 1995]. However they lack functional reusability since the component functionality or the business logic is embedded in component implementation.

Therefore there is a need for a new model driven approach which allows software developers to develop *components as logical and abstract model artefacts* that are independent from technology, yet also includes the facility to easily transform into model and framework specific components, and are therefore able to be implemented using a target technology based on the framework.

This may be achieved if the developer can design the system using a component based design model and each component:-

- represents a functional concern of a particular business, for example order management,

- explicitly declares the required and provided cross-cutting concerns, for example transaction management,

- explicitly declares the required framework environment, for example instance management support, distributed object support,

- explicitly declares the required and provided data objects, for example order object, order item object,

- and most importantly contains information about the above data as meta-data of the component, so that tools can be used to automate the processes such as analysis of the design, and code generation from the component model.

To summarise, each component should be a self-contained, specific level component. This is the main aim of the Attribute based Component Design (AbCD) approach.

4.2 Attribute-based Component Design (AbCD) approach

This section introduces the main focus of this research, which is the Attribute-based Component Design approach. Using the combination of component development principles, model driven development approaches, AOP principles and attribute-based programming, it allows developers to construct components as model artefacts that are reusable, technology independent, and yet enriched with context-based attributes. This allows the components to be easily transformed into enterprise level business components with a target technology.

As shown in Figure 18, the aim of many model driven approaches proposed in the literature is to provide an abstraction over different platform specific standards and technologies. This is done by allowing the developers to construct platform independent models and by providing code generation templates or wizards to perform model transformation to get platform/technology specific models.

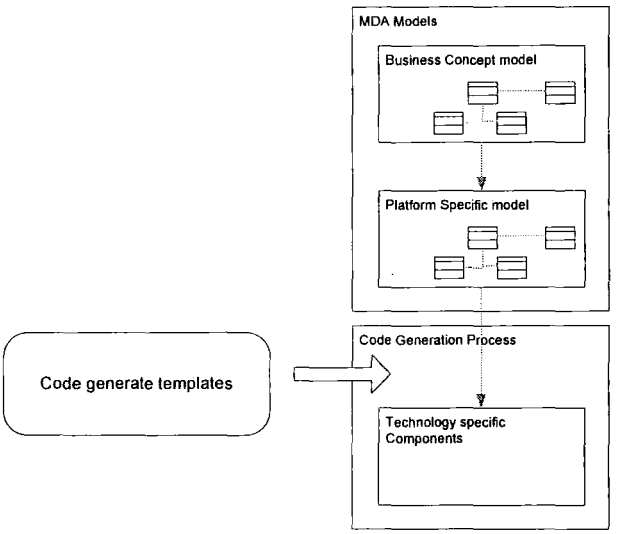


Figure 18 Generic model driven development processes

The AbCD approach differs from other model driven approaches. Most model driven development processes describe how platform independent models, such as business concept models can be transformed into platform specific models. These models can be then transformed into platform or technology specific code using transformation or code generation tools, such as the processes described in [Hubert, 2001; Kleppe, Warmer et al., 2003; Mellor, Kendall et al., 2004]. As described in the literature survey, there are many current research groups that focus on constructing code generation frameworks based on

MDA with code templates. This include tools such as AndroMDA, iQgen, ArcStyler, and Mia-Generation as listed in [Code-Generation-Network, 2006].

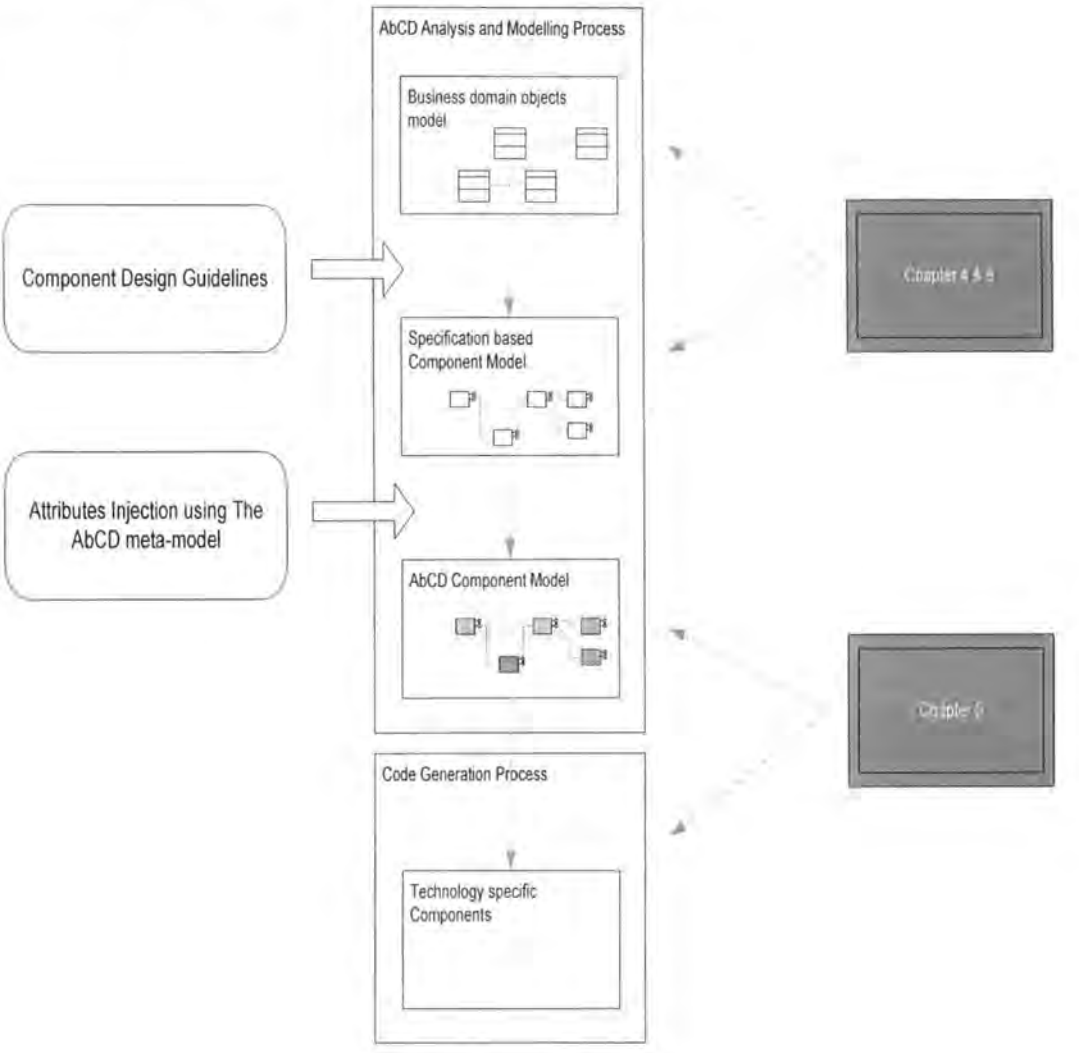


Figure 19 AbCD approach showing the modules and development artefacts

Figure 19 depicts the overall AbCD development process. The AbCD approach introduces the "***All Components***" development method. The method encourages the software developer to view all aspects of the design as logical and abstract components. The identification of such logical components provides an abstraction layer over how different functional and non-functional aspects of the design can be mapping the implementation components. The detailed description the 'all components' method is presented in Section 4.2.2.1.1 as part of Component Design Guidelines (CDG).

The focus of this research, i.e. the AbCD approach, is not to create a code generation framework for MDA. To achieve the aims described in the previous section, the following principles are introduced when modelling using the AbCD approach.

1. Identify the relationships between different business requirements based on component interaction using well defined interfaces;
2. Create a framework that allows developers to define abstract level business components. Using the technology dependency injection approach, these components can then be configured and implemented to a specific framework, without altering the component source files.
3. Identify reusable components. They can then decide on the possibility of applying 3rd party COTs as well as building in-house components;
4. Identify non-functional requirements that are overall system concerns to be resolved for all components, such as security, logging, and activation, etc;
5. Construct components that are focused on aspects, contexts, abstraction and composition, and;
6. Build a reusable component model repository.

To achieve the principles described above, the AbCD approach includes three main modules. They are:-

- **Component Design Guidelines (CDG):** This is to support the model driven development by providing design guidelines for developing component-based systems. However the guidelines do not enforce a new development process, but enforce constraints when specifying component design. It can be regarded as a "***non-invasive***" process and it is based on the design principles of all components method.
- ***The AbCD meta-model*** : This meta-model allows designers to construct UML classes as AbCD component models. Each component includes a set of meta-data as the ***Context based Attributes*** of the component. The meta-data improves the component composition and reusability.
- **The AbCD Tool Suite:** This is the realisation of AbCD approach to allow developers to practice CDG and apply the AbCD meta-model.

These three modules form a package to support and realise the six aims identified above. Figure 19 depicts the development artefacts that can be produced by applying the modules. These three modules define the scope of this thesis.

The description of the AbCD approach spans three chapters. The CDG are described in the following sections of this Chapter. AbCD meta-model is addressed in Chapter 5 and the tool suite is described in Chapter 6 as the implementation of this research.

4.2.1 A simple example: designing a simple Bank application using UML

Before discussing the Component Design Guidelines (CDG), a simple bank system example is addressed here hypothetically to illustrate how the approach can be applied.

Hypothetically, based on the requirement analysis, the following table summarises the main requirements.

Functional requirements (Vertical concerns)	Non-functional requirements across the whole system (Horizontal concerns), contractual and programmatic requirements
- Customers should be able to check balance/withdraw cash.	- Role based security for assessing the system.
- Customers should be able to use the Web client to view balance, transfer funds, apply loan applications and modify personal details.	- Transaction management for accounts and loan processing.
- The system should be able to process loan applications.	- Fast response time for processing of accounts. The system should be able to process at least 100 transactions per second.
- A bank staff should be able to open/modify/close accounts, and add/withdraw/transfer funds for customers using a GUI application.	- Persistence storage with recovery facilities.
- Additional Business constraints such as Maximum loan amount calculation rules, Customer eligibility rules, etc. should be able to be applied.	- System integration with an external credit checking system.
	- Requires a client server based system with web client for customers, GUI application client for bank staff.

In this case study, the functional requirements are not important. Using a UML class diagram, Figure 20 shows an example static structure of the proposed bank example that includes architecturally significant parts of the system. The modelling of the system using static structure and dynamic behavior diagrams provide a way of grouping functional concerns to form the design of the system. A UML static structure model, which can be represented with a UML class diagram, includes model elements such as classes, interfaces and packages. It also includes relationships amongst model elements which include association, generalisation

and dependency. By using a UML modeling tool, it is possible to construct models that use ***inheritance and association views***, and/or ***class dependency views***, as shown in Figure 20. This research explores new types of dependency views that the UML modeling approach does not focus on. For example, Figure 20 also shows that the **ICustomerManager** interface references (i.e. imports) the **Account** class by showing a dependency relation. UML dependency relations are more concrete and direct, however, there is a need to define more abstract dependency relations in order to construct components. For instance, it should be possible to define that the **Customer** and **Account** classes will depend on a logical component (i.e. a persistence service component, in this case) that provides persistence, without having to define detailed implementation technology or framework. In this research, such a dependency relation is referred to as a ***Service Dependency*** relationship. The evaluation described in Chapter 8 shows that defining an abstract dependency when modeling is important to construct feature rich components that are portable and self-reliant.

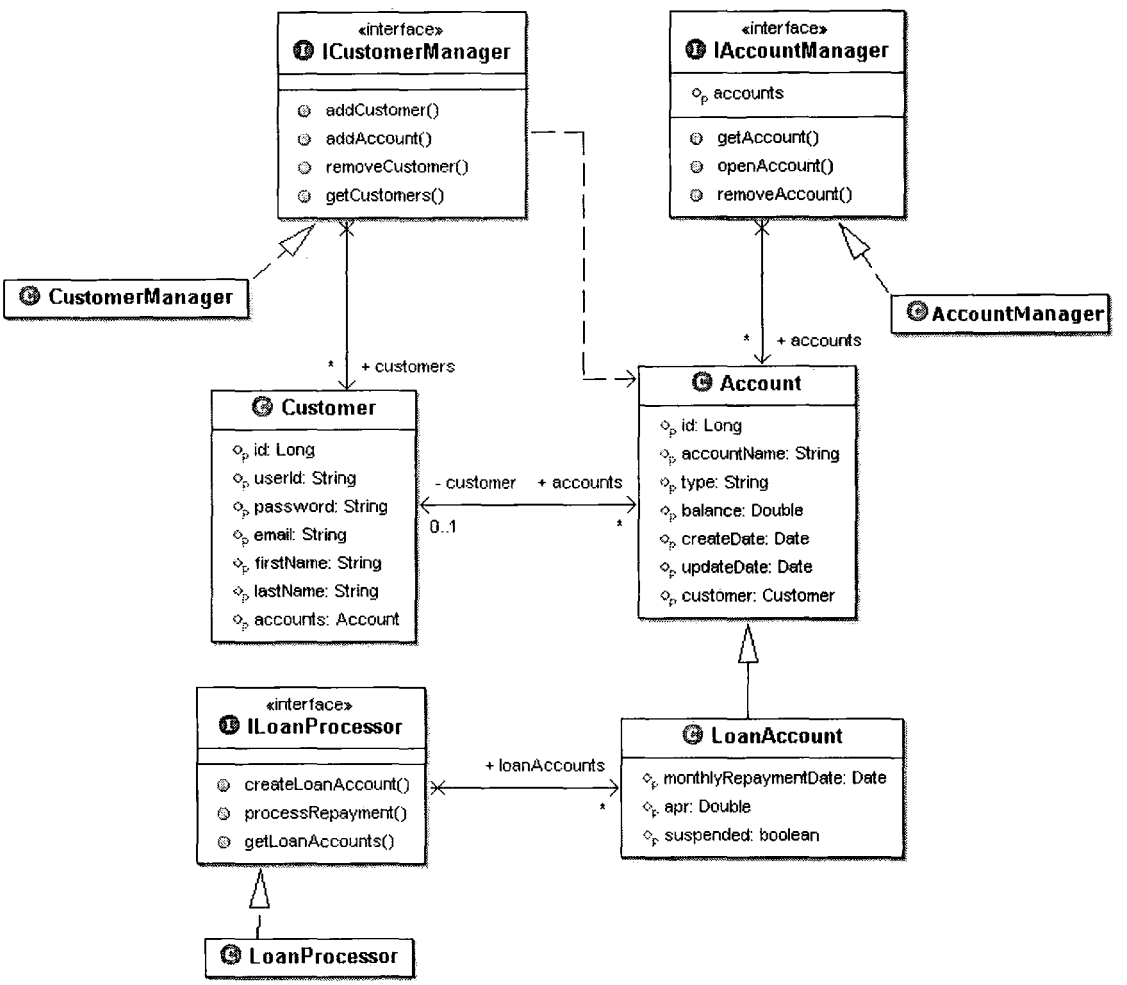


Figure 20 An example Business Concept Model for a Bank

The UML class diagram presents only partial information about the system. As presented in the literature survey, OMG also provides the Object Constraint Language (OCL) that can be

used to express formal and concise information about functional and business constraints such as pre and post conditions and invariants of classes and methods.

For instance, to add an invariant constraint that a customer's password must be at least 3 characters long, the following OCL constraint can be added:-

```
context Customer
init : self.password.length => 3
```

The constraints written using OCL can be added to UML model elements for documenting models that are more precise as well as being able to input more complete models into automation tools for source code generation. These additional constraints make the model more semantically rich and enforce the concept of Design by Contract (DbC) [Meyer, 1994].

Using the MDA concept and the code generation tools listed in Section 3.2, the business concept model of the bank example shown in Figure 20, which is also a platform independent model (PIM), can be transformed into a platform specific model (PSM), such as a J2EE model illustrated in Figure 21. In this model fragment, the diagram shows that the **IAccountManager** interface and the **Account** Class are adapted to fit within the J2EE platform. It also shows how extra J2EE specific dependency classes are added. Each class and dependency relationships are marked with J2EE specific stereotypes to reflect the model.

With the support of MDA concepts, there are many model transformation tools currently available that can transform PIMs to PSMs. In the case of the example above, these tools can be used to automatically generate a J2EE specific model from a bank concept model, using configuration settings and templates. Furthermore, these tools also provide code generation facilities, in this case, to be able to automatically generate Java code to be deployed in an EJB container.

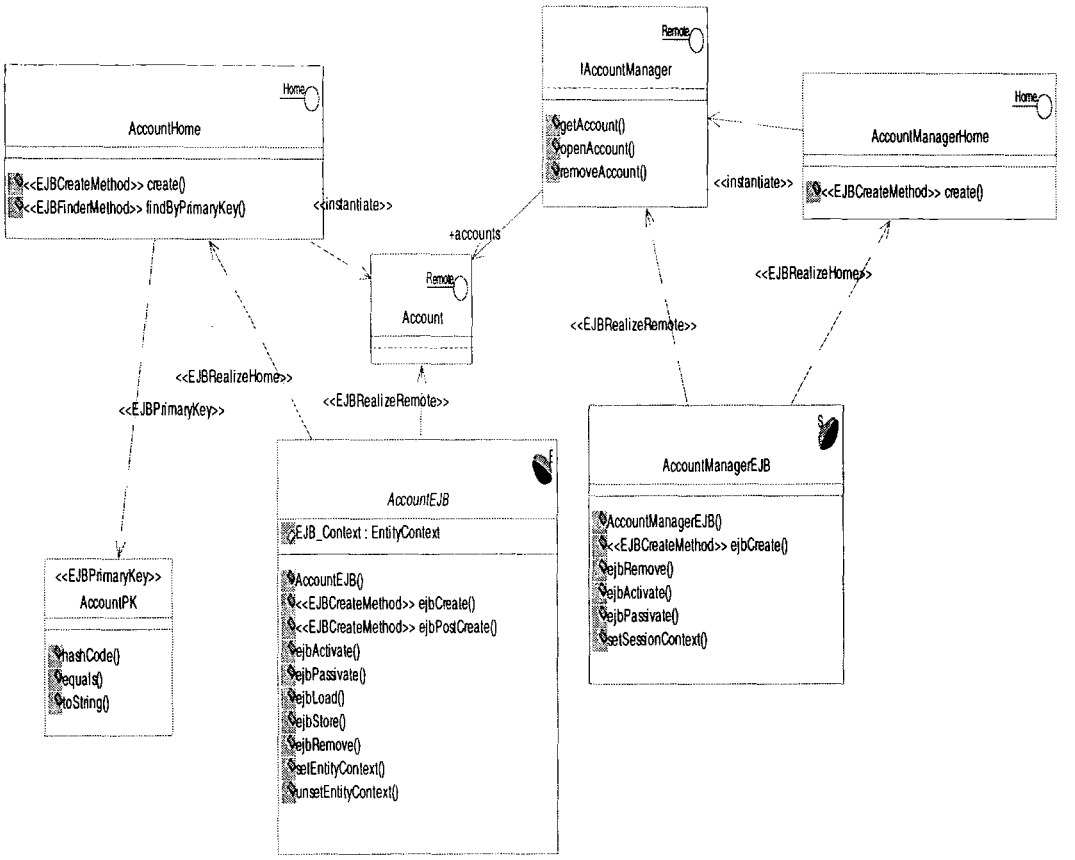


Figure 21 A J2EE specific model for the Account Manager (From Bank example)

4.2.1.1 Analysis on the Case study

The separation of PIM from PSM provides a new abstraction layer to facilitate the reuse of business models and the creation of model repositories. Component technologies such as J2EE and .NET provide facilities to implement non-functional aspects such as persistence and transaction management. Accordingly, the modelings of such aspects are only illustrated in PSM and not in PIM. Hence, the PIM can be largely different from PSM. This is one of the fundamental aspects that make the use of MDA limited. In other words, models are used only for understanding and sharing of business concepts, rather than as a development artifact.

The end product of the MDA approach is the generated code, in this case for the AccountManager and CustomerManager J2EE components, which are based on the PSM. In these components, business logic code is injected and merged with J2EE technology dependent code to provide non-functional aspects and system aspects, which make the components hard to reuse.

Component Design Guidelines (CDG) presented in the followed section shows how platform independent models can be synchronized with platform dependent models and code to minimize the need for transformation, hence increases the possibility of reusing with other platforms or technologies.

4.2.2 Applying Component Design Guidelines (CDG)

The AbCD approach introduces CDG for constructing components. CDG include a simple AbCD meta-model to construct components as modeling artefacts. The CDG is based on different approaches proposed by various researchers in the field of AOP and MDA and Dependency Injection Pattern [Harrop and Machacek, 2005].

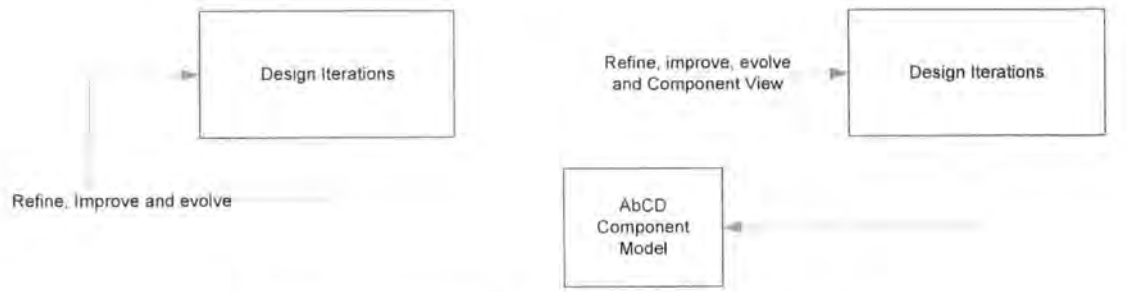


Figure 22 Applying AbCD with non-invasive approach

The CDG can be applied with a non-invasive approach. It does not impose a tight process. It is intended be applied in every design iteration as depicted in Figure 22. The CDG proposes two modelling processes for every iteration, i.e. ***Component Identification*** and ***Component Construction***. These two processes provide a new AbCD Component model with an additional component dependency view to the design. For instance, as shown in Figure 23, it can be applied using the Rational Unified Process (RUP) [Rational, 1998] (left) or an eXtreme Programming (XP) process (right).

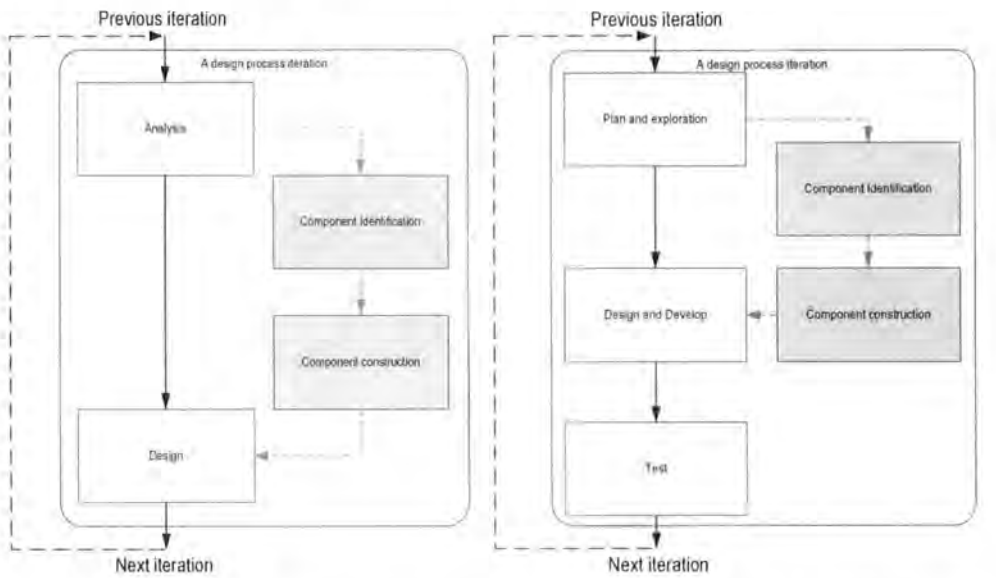


Figure 23 An expected design process iteration (left - RUP process, right – XP process)

4.2.2.1 Component Identification and Component Dependency Identification

The CDG introduces ***component identification*** and ***component dependency identification*** guidelines. Referring back to the bank example presented in Section 4.2.1, the business concept model shows that there are three functional aspects: account management, loan management and customer management. Component identification is used to promote the division of functional aspects into self-contained logical components. The system should be formed by component composition using interfaces rather than object inheritance.

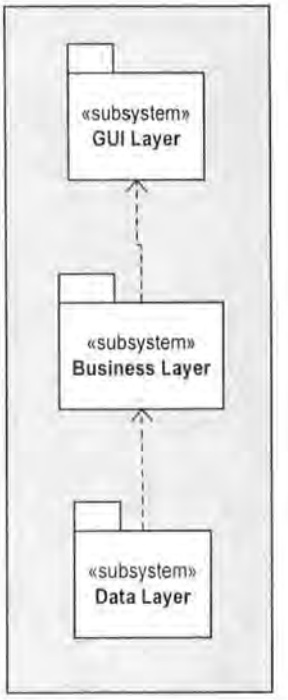


Figure 24 Partitioning the system based on 3-tier architecture

Figure 24 shows a typical high level package structure using 3-tier (or layer) architecture. The separation of the system into such packages promotes the isolation of the user interface modules from the business and data modules, thus increasing the possibility of reusing the system. However from the CBSE point of view, it is necessary to identify, and most importantly, to produce a new view of high level components that partition the system.

The CDG's first stage of the proposed design process is to identify logical components to partition the system from a static design model such as UML package or class diagrams. This process is similar to the UML modeling technique and process proposed by [Cheesman and Daniels, 2000]. However the main difference is the identification and recognition of components to support non-functional requirements. In other words, in CDG, all non-functional and cross-cutting concerns are treated as *first class* requirements.

Based on this research, the following guidelines are added to CDG when identifying potential logical components:-

Identify logical components to accommodate non-functional requirements: Although additional components for non-functional requirements may not be required to be implemented explicitly, the declaration makes the developers aware of the non-functional requirements that may impact on the architecture of the system. Most of the non-functional requirements such as transaction support, persistence, logging and traceability are most likely to have ready-made frameworks and components to be reused. This is an important aspect of the CBSE because most of the modern implementation models such as COM+/.Net and J2EE have rich features that are attribute-based for supporting enterprise level non-functional requirements when constructing the components. However there may be other non-functional requirements such as performance, reliability, and availability will not be supported by frameworks. Therefore it is possible to identify these aspects as logical component but the designer has to explicitly specify how these can be achieved by a technology when implementing the design of the system.

Identify component partitioning points for each component: The division of the system into logical and coarse-grained components promotes a Service Oriented Architecture (SOA), which is addressed in the literature survey. The developers can also manage complexity by defining clear-cut interfaces and by building the system with component composition and configuration.

Identify reusable system parts or subsystems as logical components: The declaration of logical components allows the developers to consider and identify reusable components early in the development. Developers can decide on reusing 3rd party COTS as well as in-house components.

For the bank example, the three logical components identified are 'AccoutManager', 'LoanProcessor' and 'CustomerManager'.

It is also noted that there are non-functional aspects such as, to the provision of persistence for Customer and Account information. Transaction management is also to be provided for the processing of accounts, such as transferring funds, and security for accessing Customer and Account information. During development, the model needs to be transformed to add dependency on components providing non-functional aspects. There are also system aspects such as, monitoring performance and logging.

Figure 25 shows the high level logical components identified. The idea is to form logical course-grained components based on requirements. More fine-grained UML class models and behavior models can then be added to these components.

These components identified are different from the concept of 'component' identified in the UML specification, which represent the deployment components during the implementation phase of the development. It is the designer's choice to make explicit or implicit logical components for each non-functional requirement. This phase does not introduce new methods or techniques for capturing business requirements into class models. The division of the system into logical components at the early stage of the development is a significant change to traditional OOAD.

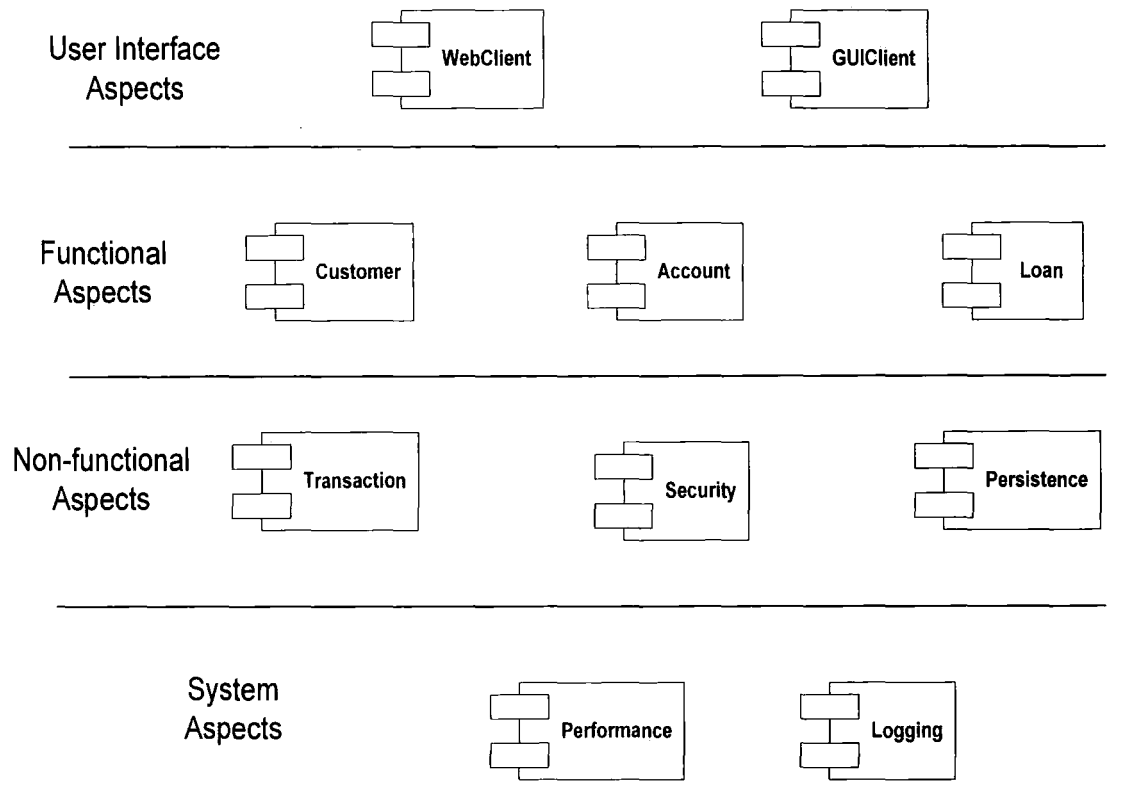


Figure 25 Component structure for Bank application

The grouping of the system in this way is similar to partitioning of a system using subsystems and packages in UML. However, this organization promotes the identification of component dependencies based on aspects rather than structure. It is important for a component developer to focus on component dependencies at an abstract level.

4.2.2.1.1 The "All Components" method

All components method is proposed by CDG. Firstly, all objects in the design must map to a logical component. In Object Oriented design, dependency between objects is achieved by inheritance (sub-classing) or association. In component based development, the designers are

encouraged to avoid inheritance and dependency is formed using interface composition. This is to avoid fragile base class problem identified and explained in [Szyperski, 1998].

However avoiding implementation inheritance is sometimes impossible. Using CDG, logical components are formed based on different aspects, either functional or cross-cutting. The method states that the designers should avoid implementation inheritance across logical components identified when mapping the classes in the design. This ensures that only interface compositions exists across components, and thus across aspects. This means that an inheritance relationships should only occur with a particular logical component. This promotes Design by Contract at specification level.

4.2.2.2 Component Construction

With the introduction of ***logical dependency injection***, which is described in next section, it is possible to specify how functional components are dependent on non-functional components. For example, the IAccount component depends on the IPersistence component at a logical level. However, when implementing persistence for the IAccount component using a particular framework, the Account object within the component might have to either statically bind with the component that provides the functionality or dynamically bind at runtime. For instance, to enable persistence for the Account and Customer objects using J2EE container managed persistence, these objects must implement the EJB entity bean interface **javax.ejb.EntityBean**. Therefore the Account EJB entity bean can be serialised by the J2EE container using the configuration settings from deployment descriptor. The transformation of PIM objects to J2EE objects introduces additional dependency. In this research, frameworks such as J2EE and CORBA/CCM, are regarded as ***Heavy Weight*** frameworks. This is because they enforce dependencies to the components over the framework. It is also possible to model components to use ***Light Weight*** frameworks [Harrop and Machacek, 2005]. Light weight frameworks are generally based on the concept of Inversion of control IoC and the dependency injection pattern [Harrop and Machacek, 2005]. The following sections describe the construction of components for these two different approaches.

Based on these different framework approaches, a new meta-model, called the ***AbCD meta-model***, is constructed as part of the CDG to facilitate the construction of components. The detailed specification of the meta-model is described in next chapter.

As shown in Figure 26, ***AbCD component models*** can be constructed based on specifications defined under the Meta-model, in other words, the component model for the application is an instance of the Meta-model.

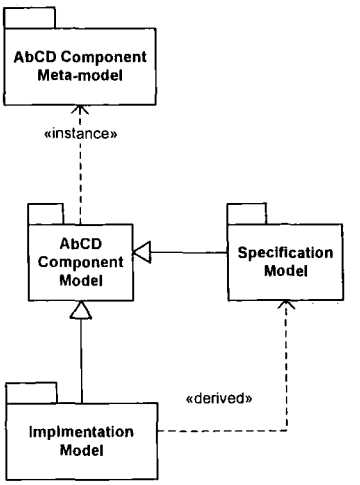


Figure 26 an AbCD model overview

The CDG focuses on modelling the components at two levels, namely the specification level model and implementation level model. In other words, a designer can construct a specification and/or an implementation model as an AbCD component model. The specification model is designed to represent a system that is independent of technology or any implementation specific model elements. The implementation model is derived from the specification model to include platform and implementation specific attributes to model elements. This separation is necessary to make the specification model as an abstract model to improve reusability of the component model.

However due to the lack of support for adding non-functional constraints to UML models, such as *response time* in this example, many researchers, from the area of AOP, are proposing different solutions to facilitate non-functional aspects to UML models, as presented in [Suzuki and Yamamoto, 1999; Grundy, 2000; Clarke and Walker, 2001; Stein, Hanenberg et al., 2002; Rashid, Moreira et al., 2003]. It is observed that the methods used in these articles can be grouped into two as follows:-

1. The use of UML stereotypes to identify and differentiate UML model elements as special cross-cutting elements, as presented in [Stein, Hanenberg et al., 2002].
2. The extension of UML meta-model to include special woven and aspect classes to current model elements, as presented in [Suzuki and Yamamoto, 1999].

CDG focuses on capturing functional and non-functional requirements to component design by extending and adapting these existing AOP extensions to UML modeling.

The CDG process does not introduce a new process model, but aim to complement existing model driven Object Oriented analysis and design processes such as RUP or Catalysis approach presented in [Rational, 1998; D'Souza and Wills, 1999]. However it focuses on how to model components using Object Oriented Methods.

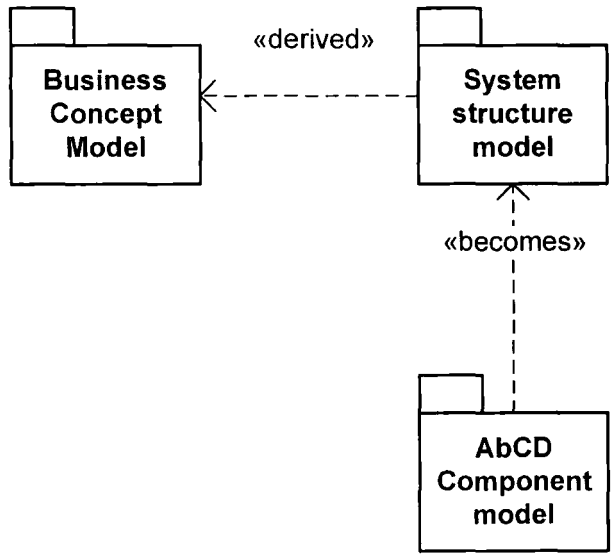


Figure 27 Logical component modelling using CDG

Accordingly, the CDG process does not focus on an Analysis model or Business model, but on the static structure model of the system, in other words class diagrams. In RUP, these analysis models capture the current process and structure of the business, and they are used by developers to understand and share the concepts. The result of applying CDG is to derive AbCD component models from system static structure models such as UML class diagrams that describe system design and functional requirements. In other words, system static structure models ***become*** AbCD component models as shown in Figure 27.

4.2.2.3 Summary of the CDG

To summarise, the CDG is based on the following modeling principles:-

- All non-functional and cross-cutting concerns are treated as *first class* requirements. This means that such concerns are addressed explicitly when a transfer is made from requirement analysis to design of the system.

- Modeling to build "components". Components and component interfaces are also treated as *first class* entities rather than classes and objects. Business operations are grouped into logical components rather than class models.

- Modeling for "reuse". It is well documented that reusing implemented components is difficult [Garlan, Allen et al., 1994]. This is because the code is written to work with a specific platform or system and it embeds deployment model specific and technology specific parts within it. CDG focuses on building components as abstract modeling artefacts for reusability, yet include attributes for easy transformation of these artefacts to deployment specific components.

- Modeling with "abstraction". The guidelines ensure the constructed component artefacts are a level above today's modern platform dependent components such as .NET assembly, or EJB deployment components. However the components should be easily transformed with standardised MDA mapping tools.

- Modeling for "implementation". In most cases of model driven development, models do not reflect implementation. The guidelines enforce the construction of logical component models that reflect physical components.

Chapter 5 AbCD Meta-model

5.1 Introduction

This chapter details a specification for a meta-model. The meta-model, known as the AbCD meta-model, is presented here to support the modelling of software components. The meta-model combines the concepts of interface-based component composition, aspect-oriented programming, and attribute-based meta-modelling. The details of these concepts are discussed in Chapter 3 as part of the background literature.

This chapter addresses how component models can be produced based on the AbCD meta-model. It also describes how the models can be presented in various formats or representations for different stakeholders of the development. This is an important issue that bridges two different usage of MDA, i.e. building models for sharing the understanding of the concepts and building models as development artefacts, to be used in implementation of the component. The detailed discussion on different types of artefacts is made in the next section. This chapter starts by discussing the scope and target of the models to be produced. It also describes the focus areas that the AbCD meta-model is targeting for improvement when constructing components. This meta-model is to be applied as part of the simple process identified as Component Design Guidelines (CDG) in Section 4.2.2.

The AbCD meta-model specification is projected using a MOF model and an UML profile. The profile is compliant with the Model Driven Architecture (MDA) introduced by the OMG. This chapter then outlines the usage of the meta-model using the bank example presented in Section 4.2.1.

5.2 The modelling artefacts of MDA

Before discussing the meta-model, it is important to address the scope of the models to be produced using the UML meta-model. From the literature, Dobing has done a survey on the usage of UML to 182 respondents (171 UML users and 11 partial users) [Dobing and Parsons, 2006]. The findings showed that only 6 projects used UML from 27 projects (only 23%) involved by respondents. The following findings are also presented.

- "Only Class Diagrams are being used regularly by over half the respondents, with Sequence and Use Case Diagrams used by about half."
- "When asked whether the UML facilitated communication with clients, 55% said it was *at best* moderately successful"
- "Class Diagram (73%) is the most frequently used technical description, followed by Use Case Diagram and Sequence Diagram."

- "Use Cases Narratives (87%), Activity Diagrams (77%) and Use Case Diagrams (74%) are the preferred means with regard to client involvement."
- When asked about "the reasons for not using some UML components, 50% said that Class Diagrams were not well understood by analysts, 48% said that Activity Diagrams were not well understood by analysts."

To make some observations from these findings, the usage and involvement with UML in projects are very low. Also the class diagram is the most used but the lease understood. UML is best used as a communication medium for sharing concepts but not as a model presentation of the system for implementation.

As a part of this research, a small questionnaire based survey was carried out to obtain a different general perspective. The results shown here were collected from a section of a large software development organisation, involving 8 developers, 2 senior software engineers and 1 configuration manager. This main goal of the survey is to gather a view from practitioners about:-

- the ***role*** of UML and the use MDA in software development lifecycle,
- the use of ***graphical and other notations*** of UML models in software design,
- and the use of UML for ***platform independent or dependent design***.

The rationale behind this survey is to verify the current problems of the MDA approach and to validate the concepts, which are identified and added to the meta-model specification, can be beneficial to practitioners. Furthermore, it is to study an overall role of MDA when producing various development artefacts, and hence to derive a focused specification for the meta-model.

- The following questionnaire was made regarding the role of UML and the use of MDA.

  Models are made for different purposes or ***roles***. To be more specific, models can be used for representing the conceptual domains, business process, data representation, and structural design of the system to be built. From the MDA literature, models are made mainly for three different target artefacts.

  Analysis model artefacts: They are built as analysis model artefacts. This is to gain more understanding of the problem and system to be built.

- ***Design model artefacts***: They are built as design model artefacts. This is the actual representation of the system to be built. The survey result shows that if the MDA approach is applied, only the analysis model artefacts are mainly produced (30%),

and not the system design artefacts (only 5%). The following table describes the usage of UML in design by these developers.

Question	Answers from developers (out of all projects they designed)
Do you use UML modelling for understanding of concepts?	30 %
Do you use UML models as primary development artefacts? (i.e. do you use MDA approach in software development?)	5 %
Do you use UML modelling for high level design and architecture of the system only and not for the detailed design of the system?	15%
If you use UML modelling for system design, other than class diagrams, do you use other UML modelling diagrams such as statechart and collaboration diagrams?	35%

The results show that as the coupling between application components that provide system functionality and library components that provide non-functional requirements increases, UML lacks the ability to encapsulate the library components to higher level abstractions within the design to reduce complexity. One way to resolve the encapsulation problem is to introduce meta-information about non-functional requirements within the application component model elements and to omit the library component model elements in the system design. This aspect has been added to the meta-model and more elaboration is made when presenting the meta-model specification in Section 4.3. Another important aspect identified was UML lacks the ability of the relation between logical components and detailed implementation components. In other words, it is difficult for a designer to trace how logical components identified are transformed into detailed physical components used in the system.

Pattern model artefacts: Models are used to represent patterns. These model artefacts can be said to be part of the analysis model. From a software engineering point of view, patterns are reoccurring common problems and solutions. Models can be used to describe analysis patterns.

> "Analysis patterns describe solutions to common problems found in the analysis/business domain of a system." [Hay 1996; Fowler 1997; Ambler 1998a]

Models are also used to represent design patterns [Gamma, Helm et al., 1995]. The former focuses on solutions to a particular system to be built and the latter is used to describe generic design problems and solutions in software development.

- The following questionnaire was made regarding the use of different modelling notations in MDA.

Questions	Answers from developers (out of all projects they designed)
Do you build UML models to vistualise program structure as graphical model only?	40 %
When using UML as a modelling language, do you use other forms of representation of a UML model, such as XMI?	5 %

The result suggests that UML modelling artefacts are produced mainly as graphical models. In other words, model artefacts are mainly represented with the form of graphical notations. As already described in the literature survey, OMG provides UML modelling constructs for building UML models and MOF modelling constructs for building other domain specific models. These constructs provide graphical notations to build models as graphical diagrams, such as UML class diagrams, and UML collaboration diagrams. However graphical diagrams are useful as visualisations that support program understanding.

Accordingly, graphical modelling is more appropriate for documentation and analysis of the domain, and hence as analysis models. However, when designing the system using modelling, i.e. when building design models, it is important to represent models in various formats. Thus tools can be used to ***generate***, ***analyse***, and ***refactor*** the implementation code.

- The following questionnaire was made regarding the uses of different modelling notations in MDA to MDA users.

Questions	Answers from developers (out of all projects they designed)
Do you build Platform Specific Modelling as well as Platform Independent Modelling?	10 %
Do you use UML profiling approach to extend the model for a particular platform or framework?	2 %

The result implies that if a model driven approach is used in software development, only 10% of models are platform specific models in small projects. Models can be made platform independent or platform specific. As part of MDA, OMG has also introduced the notion of PIM and PSM. It is possible to add meta-data about platform or framework specific information to a PIM. One way to adding the meta-data is to apply a UML profile of a particular platform to PIM. Constraints about the platform

and its semantic information are captured by using stereotypes that are applied to model elements. Tools can then be used to generate platform specific code. The detailed discussion on OMG's PIM and PSM was made in Section 3.2.1.

This survey on the use UML and MDA shows that, when a model driven approach is used in software engineering, mainly analysis model artefacts and pattern designs artefacts are widely produced in comparison with system design model artefacts.

The main aim of proposing this meta-model is to widen the use of a model driven approach and to produce design model artefacts that represent software components. Figure 28 shows an overview of a simple modelling workflow. Design models can be derived from Analysis models.

However, in a component development environment, it is difficult to build a complete design model of a component because it generally depends on a particular framework or a wiring standard that the component is based on. Therefore it is necessary to construct an abstraction representing many different component frameworks. This is achieved by reviewing common concepts from different technologies for each of the frameworks.

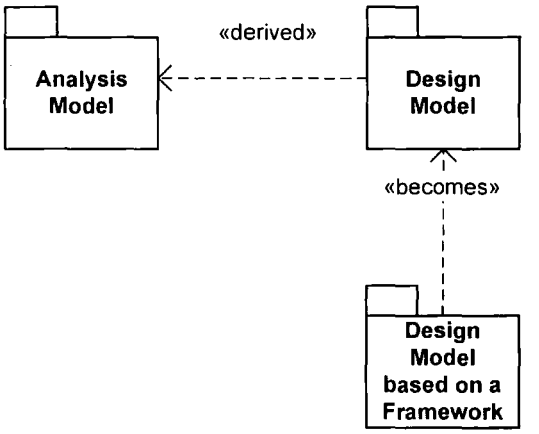


Figure 28 An overview of the modelling workflow

In this way, a model that shows a generic pattern of the common features and an abstract architecture of the currently available component frameworks can be produced, and hence will be able to produce a generic component meta-model.

5.3 AbCD Meta-model and a AbCD UML profile

This section introduces the Attribute-based Component Design (AbCD) Meta-model. This work is carried out to define a meta-model that can provide a standard means of modelling business components using UML profiling and hence to allow integration with UML tools. It

focuses on supporting developers to construct design model artefacts, as apposed to analysis model artefacts. The meta-model can be used to model business components at a higher abstraction level. The central concept of this meta-model is the addition of ***attributes*** to model elements. Using the meta-model, this section shows how a component model can be constructed as ***an abstract model,*** which focuses on functional aspects and as well as encapsulating all non-functional aspects that should be provided by the container. The meta-model consists of various attributes which describe component requirements that should be provided by the component container. In other words, the attributes added to model elements are: information about the required and provided services by the component, required container features needed by the component and, and required context information. These attributes can be regarded as meta-data that can feed into a generic business component model to enrich information about the required environment needed by the component.

The AbCD meta-model imports the UML 2.0 Superstructure meta-model to provide a standard way of presenting the semantics of the component model, as shown in Figure 29.

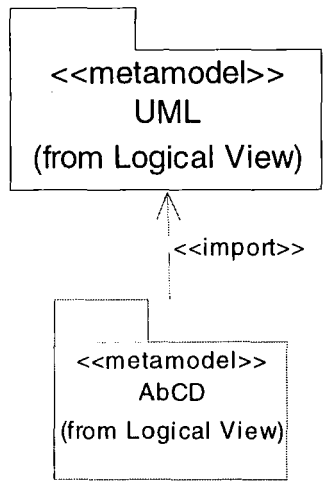


Figure 29 Extending the UML meta-model

The UML model elements that are extended from the UML specification are referred to as *UML meta-classes*. The AbCD Meta-model extends four main meta-classes from the UML specification. They are as follows:-

- Interface
- Class
- Association
- Dependency

Figure 30 depicts the abstract model of the AbCD meta-model concept. The meta-model is expressed using the graphical notation specified in the MOF syntax specification. This meta-model introduces ***new types of meta-classes*** that extend standard UML meta-classes from the UML specification. They are :-

- **AbCDComponent meta-class**
- **AbCDServiceComponent meta-class**
- **AbCDDataComponent meta-class**
- **AbCDComponentAssembly meta-class**

The diagram also shows how these UML meta-classes are extended. It is important to note that the extended meta-classes add constraints that represent extra semantic information attached to the extended element.

This section also describes the use of the UML 2.0 profiling method to present the AbCD meta-model. Using UML 2.0 profiling, it is possible for any modeling tools that support UML 2.0 profiling to be able to apply the AbCD meta-model. As described previously, the AbCD meta-model introduces new types of meta-classes that extend UML meta-classes. Using the UML profiling approach, AbCD meta-model is formed as a new UML 2.0 profile, called ***AbCD Profile***. The profile consists of a set of new stereotypes. A stereotype can be regarded as a virtual meta-class of the AbCD meta-model. The meta-class (and hence the stereotype) depends on the UML meta-class that it extends. Hence applying a stereotype to a UML model element implies that the model element becomes associated with the AbCD meta-class that the stereotype represents.

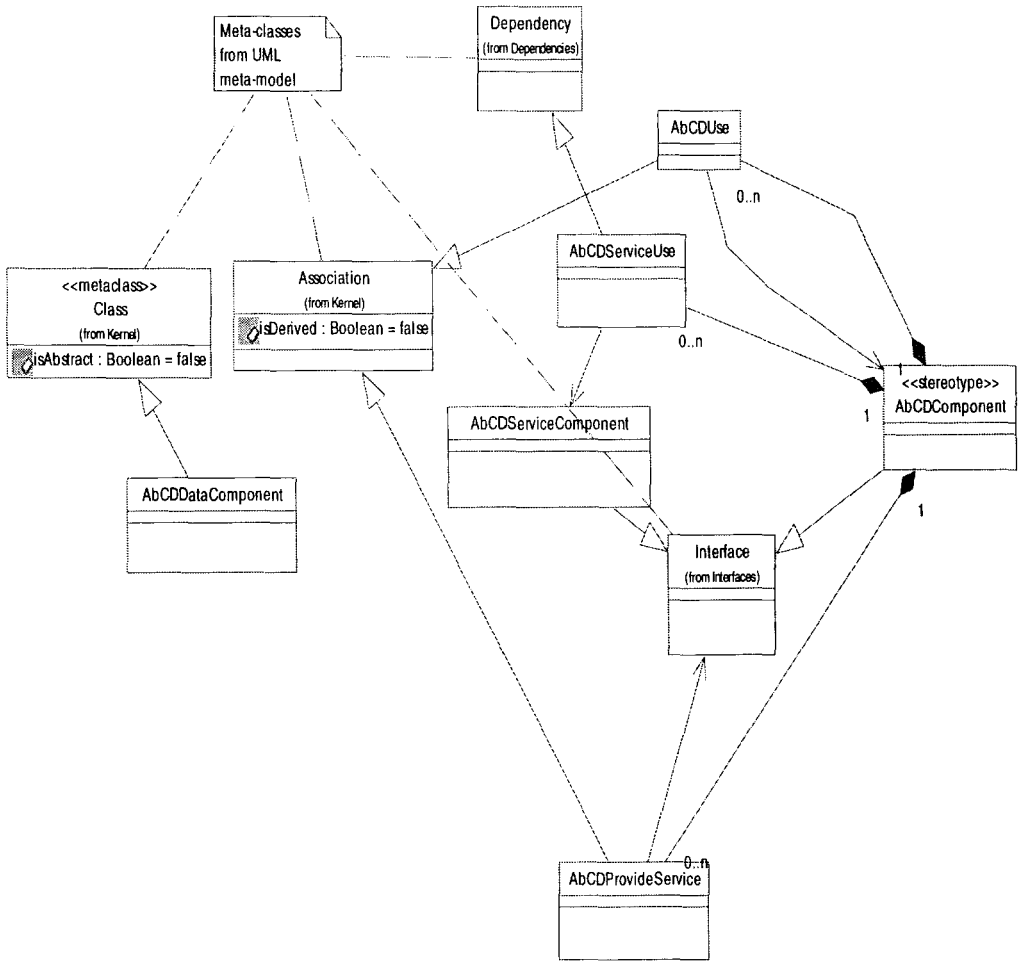


Figure 30 AbCD Abstract Model

Using UML profiling, a stereotype may have a set of attributes, tag values and constraints. Using this facility, the AbCD profile adds constraints to stereotyped model elements using attributes that focus on providing a clean separation between component business logic, component services and other non-functional aspects of the component.

A corresponding stereotype is added to a UML model element as a visual representation of the new type of model element. The following section discusses the main goal of each type of meta-class and rationale behind the concept. Figure 31 shows an explicit model of how a stereotype is extended from UML meta-model.

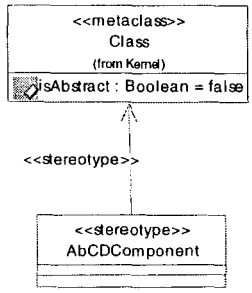


Figure 31 Model of AbCDComponent Stereotype

5.3.1 AbCDComponent meta-class

The AbCDComponent type defines a representation of an ***abstract*** component. It is an extension of the UML meta-class '*Interface*'. It must be represented using the <<AbCDComponent>> stereotype. The main goal of introducing this meta-class is to allow a designer to define a logical component with a higher level of abstraction.

A component may provide one or more services as well as require other services. To facilitate this requirement, UML 2.0 introduces the notion of ports to describe required and provided services. The provided services are presented though the use of Interfaces. An interface is a cohesive set of functionality for a particular aspect provided by the component. A component may provide one or more interfaces.

This meta-model further refines this requirement by introducing three new association meta-classes, ***AbCDProvide***, ***AbCDUse*** and ***AbCDServiceUse***. Figure 32 shows a fragment of the model from the bank example. When the designer attaches the AbCDComponent stereotype to the BankManager interface, it is regarded as an AbCDComponent and hence the following constraints apply to it:-

- It is an abstract and logical component.
- It will be deployed on a container when implemented.
- It will use interface composition if possible. Depending on the container, the component may use a dependency injection method or a service lookup method to integrate with its collaborators. It also means that the dependencies are explicit.
- The component design includes an explicit declaration of service dependencies needed by the component, using the AbCDServiceComponent stereotype declaration. The services may be provided by the container library or may use 3rd party service components.

- The component design also includes an explicit declaration of the data objects that it uses and shares. The data objects should have the AbCDDataComponent stereotype.
- Finally, the component includes a set of *contextualised attributes and corresponding values* regarding all non-functional requirements, runtime requirements, and other service requirements needed by the component.

The diagram also shows how the AbCDComponent stereotype is applied on the BankManager component. It provides both IAccountService and ICustomerService interfaces.

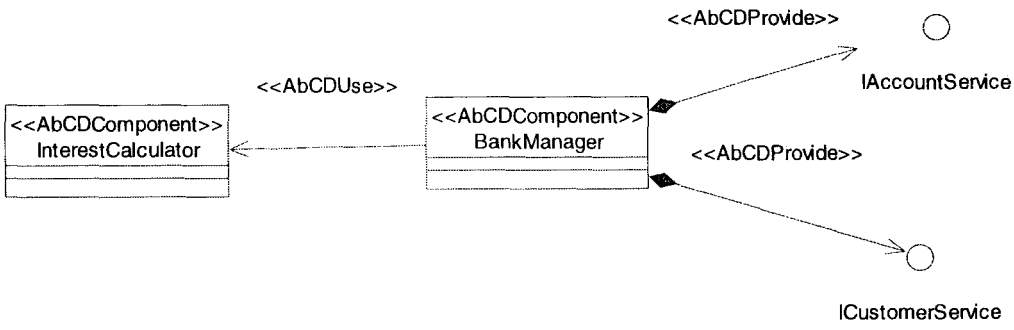


Figure 32 AbCDComponent stereotype for Bank Example

The composite association between the component and its provided interface is stereotyped with <<AbCDProvide>>. This is to enforce the contractual relationship between components. The association between the component and its referenced component is stereotyped with <<AbCDUse>>. The AbCDUse meta-class is an extension of the UML meta-class ***Association***. It defines an association between two AbCDComponents, indicating that one component references the other. The AbCDUse meta-model element is expressed using the stereotype <<AbCDUse>>.

As described previously, the main goal is to add meta-data to model elements. Hence, although the BankManager component is an abstract and logical component, the designer can fill the component with ***contextualised attribute values*** regarding the component requirements. These attributes are collected as a result of the research carried out to identify common features in component design when using component frameworks. Accordingly, these attributes do not focus on any specific technologies.

The AbCDComponent stereotype has the following attributes which add semantic information to the component. These attributes were collected from the study of the three heavy weight component frameworks and also light weight frameworks presented in the literature survey, Section 2.2.4. The attributes cover different non-functional aspects of the

component. The attributes are common to all frameworks although they may provide different implementation support.

Attributes	Type	Description
Lifecycle Management	Text	A constraint on how the component should be managed by the component container, including activation, and instance management.
Classification	Enumeration	Classification of component type. Currently supported types are: • Business model component • Desktop component • Utility component • Web controller component • Unspecified
Factory	Boolean	A constraint on whether the component requires a factory object for instantiation.
Category	Enumeration	A constraint on the relation between the component and its clients. • Session • Unspecified
Event Management	Text	A constraint on the event management service that should be provided for the component.
Remoting interface	Boolean	A constraint on which the component requires the remoting interface for distributed clients.
ActivationType	Enumeration	A constraint on whether the component should be :- • Singleton • Instance
ThreadModel	Text	A constraint on which the component is designed to be

		used as eight single treaded model or multi threaded model.

Table 6 Contextualised Attributes for the AbCDComponent meta-class

The attributes outlined above are initial minimum set of generic attributes that can influence the architecture of a component-based system and technology selection for implementation of components.

One of the most important contributions of having such non-functional properties as attributes in logical design is that it can help the developer when acquiring 3rd party components, service components and selecting technologies for implementation.

5.3.2 AbCDServiceComponent and AbCDServiceUse meta-classes

The AbCDServiceComponent type defines a representation of a logical component that provides one or more non-functional or cross-cutting services. It is an extension of the UML meta-class ***Interface***. It is presented using the stereotype <<AbCDServiceComponent>>. The main objectives of introducing a service component type in the component design are:-

- to identify components that should be provided by the container,
- to promote reusability of the service components,
- and if necessary to be able to refactor the design to separate business components from service components.

The separation of service components introduces an Aspect Oriented Programming concept to component design. The component designer should ideally define one component for each non-functional or cross-cutting aspect of the system. Hence the component design depicts not only the dependency between components, but also different aspects of the component that depend on the functionality provided by component framework.

The AbCDServiceComponent stereotype has the following attributes to add semantic information to the component.

Attributes	Type	Description
Scope	Enumeration	A constraint on which the Scope of the service should be applied. Currently supported scopes are:-

		• Method • Instance • Field • Thread
Context	Text	A constraint about the component context to be used by container. It defines the context the application needs to use the service.
Framework Support	Boolean	A constraint indicating that the service should be provided by the component container.

Table 7 Contextual attributes for the AbCDServiceComponent meta-class

The use of interfaces to describe the component services is standard practice and it is not the focus of introducing this meta-model. The main focus is on describing the required services. An important aspect of component dependency is the relation between the AbCDComponent and the AbCDServiceComponent.

The dependency relation between AbCDComponent and AbCDServiceComponent is expressed using the AbCDServiceUse meta-class. It is an extension of the UML meta-class Dependency. It must be stereotyped with <<AbCDServiceUse>>. The meta-model contains the following attribute(s).

Attributes	Type	Description
AspectName	Text	The role name of the aspect the AbCDServiceComponent is providing to AbCDComponent

Table 8 Contextual attributes for AbCDServiceUse meta-class

Figure 33 shows how the AbCD meta-model can be applied using the Bank example introduced in Section 4.2.1. The model is presented using a UML class diagram. Each stereotype is labelled with <<stereotype name>>.

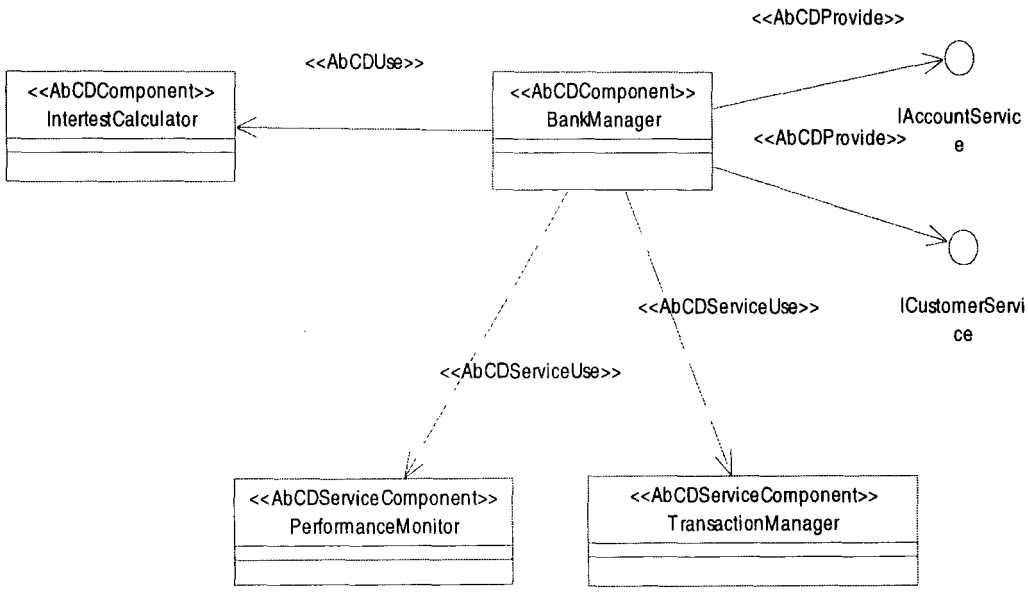


Figure 33 Applying AbCDServiceComponents to the BankManager component

The class diagram shows a fragment of the bank model, which includes the provided and required services by the BankManager component. It is an AbCDComponent model element and it is expressed with the <<AbCDComponent>> stereotype. This means that the BankManger component should be provided with a container. The component requires two services: performance monitor and transaction manager services, which are depicted with <<AbCDServiceComponent>> stereotype in the model diagram.

5.3.3 AbCDDataComponent

Another important aspect that influences the component design is the data used and shared by the component. The AbCD meta-model introduces a new meta-class called AbCDDataComponent. It is an extension of the UML meta-class ***Class***. It is presented using the stereotype <<AbCDDataComponent>>. There are two main reasons for introducing the meta-class.

Firstly, using the meta-class, the explicit representation of the component data structure can be made. This is significant for component design to be able to truly encapsulate the component implementation details. Figure 34 shows an example component diagram when applying the AbCD meta-model to the bank example. The diagram does not focus on the logical relationships between components, i.e. Customer may have many accounts, an account may have transaction history, etc. However it depicts how the BankManager component exposes three data components Account, TxHistory, and Customer.

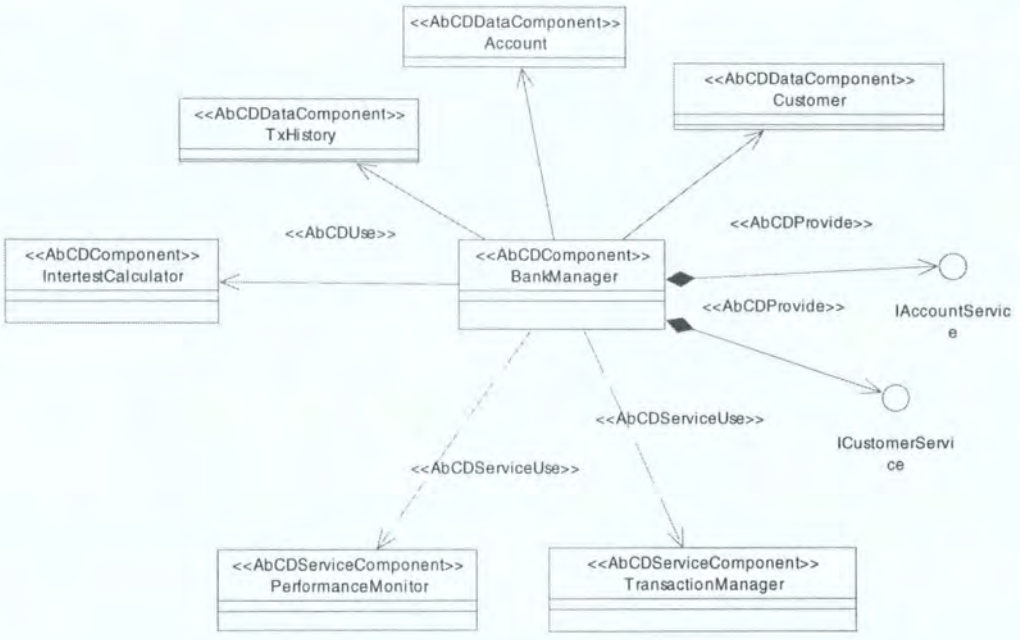


Figure 34 Applying AbCD meta-model to the Bank example

Secondly, the AbCDDataComponent meta-class is introduced to create a higher abstraction level for data access level services. With the emergence of different data access services, such as databases and XML storage services, a component developer can choose a variety of persistence methods. Each method can alter the design of the component. The AbCDDataComponent encapsulates the detailed design of each of the data services, to create data structure that will meet the non-functional constraints imposed on the system design. The AbCDDataComponent has the following attributes to add semantic information.

Attributes	Type	Description
Persistence	Boolean	A constraint on which the data component requires persistence service.
PersistenceService	Text	A more detailed description on persistence method.
Mapping	Text	The details of ORM mapping configuration.
Serialisable	Boolean	A constraint on which the data component is serialisable.

Table 9 Contextual attributes for the AbCDServiceComponent Meta-class

Referring back to the bank example shown in Section 4.2.1, the component designer for the BankManager component may add attributes about the persistence service to Account, Customer, and TxHistory data components. Accordingly, it is the component implementer's task to apply the appropriate persistence approach and technology when constructing the component.

5.3.4 AbCDComponentAssembly meta-class

The AbCDComponentAssembly component represents the physical packaging of the AbCD components. In other words, it signifies the file that contains the logical component model. It is different from the implementation component package that contains component binary classes because it denotes a ***unit of design*** for deployment rather than a unit of implementation code for deployment. It is an extension of the UML meta-class Component and is attached with the <<AbCDComponentAssembly>> stereotype to add the semantic information.

The AbCDComponentAssembly is included in the meta-model to provide a way of deploying logical components as a reusable component specification.

5.3.4.1 Unit of deployment

One of the core CbSE principles is that components are units of deployment. The component should be able to be deployed independently and also be composable to form a component based system. Generally, when deploying, the components are compiled binary components at the implementation level. This meta-class captures a component deployment at the design level. This means that components can be deployed as design models rather than binary components. This encapsulates and abstracts away components that are technology specific.

Therefore it is important to note that deployment in this case is referred to as deploying to a design model for composition rather than deploying to a system for running and using the component.

The ability to deploy components at the specification level changes the perception of component reusability. In CbSE, reusing a component is generally referred to as reusing implemented components rather than at the specification level. As already described in the literature, the reusability of the components is difficult, because the components are implemented using a specific component framework and technology. The notion of forming a deployable assembly as specification components promotes the possibility of reuse for components as a black-box unit of deployment. One potential benefit to this notion is the possibility to form a shared library of component designs as collection of

AbCDComponentAssemby packages. This is one of the aims set out to achieve in the AbCD approach.

5.3.4.2 Constructing an AbCDComponentAssembly

Logical components to be grouped into an assembly for deployment by constructing a new UML package and by applying the AbCDComponentAssembly stereotype. As shown in Figure 35, the stereotyped package can be used to represent a collection of logical components in a specification form. This means that the package is a component design assembly which can be deployed to other design.

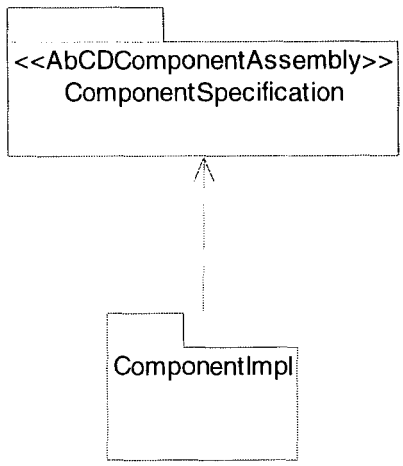


Figure 35 The AbCDComponentAssembly for packaging component specification

The AbCDComponentAssembly meta-class consists of a set of attributes. In other words, the AbCDComponentAssembly adds the following semantic information in the package as attributes.

Attributes	Type	Description
Version	Text	The version of the assembly. The version attribute enables the component to evolve separately from clients or side by side.
Shared	Boolean	This attribute defines whether or not the component is shared amongst clients or privately used within the client.
DeployInfo	Text	This attribute allows developers to describe how the assembly can be deployed. This includes the dependency required by the component by the

		deployment environment.
ImplementationMapping	Text	The component developer can use this attribute to describe the implementation related to this component specification model if any.
UML version	Text	This attribute can be used to describe the UML version used to define the component specification model.
AssemblyInfo	Text	This attribute is essential for the AbCDComponentAssembly to work. The AssemblyInfo attribute is a meta-data of the assembly that describes all the component types, interfaces and data objects required and provided the assembly. Clients can use this information to use the assembly.
Name	Text	This attribute describes the name of the assembly.
Description	Text	This attribute can be used to further describe assembly details.

5.3.5 Summary of the meta-classes introduced in the meta-model

All modern component-based middleware technologies provide similar standards, design principles as well as enterprise level non-functional services as described above. Based on such similarity, the ABCD design approach can be used to form a generic meta-model that allows the developer to produce component specification models that are abstract and independent of any implementation contexts.

5.3.6 Component Dependency View with Colour regions

With the introduction of AbCD Meta-model, it is possible to create a new ***component dependency view*** to extend the current views supported by UML. The *UML class diagram* provides different model elements to construct a static structure representation of the system design. However the UML class diagram elements are generally recognised as modelling constructs for object oriented design, and are not suited to represent logical components in component based design. It is widely acknowledged in CbSE community as described in [Heineman and Councill, 2001]. To resolve this, OMG has introduced a new notion of component diagrams in UML 2.0. As part of the literature survey, detailed discussion on MDA and UML was made in Section 3.1. There are two important aspects of the component diagrams in UML 2.0 that can be identified. Firstly, they can be used to represent mainly course-grained high level artefacts. Secondly, they can be used to represent component composition using interfaces and ports. With the introduction of different component types in the AbCD meta-model, the component dependency view adds a new perspective for AOP in component design. It also shows a low level component composition view.

In Section 4.2.1, a bank example is used to discuss how the AbCDComponent and the AbCDSericeComponent stereotypes are applied. Each service component, i.e. AbCDServiceComponent, represents an aspect or a service needed by the AbCDComponent, which is the BankManager in this case. The BankManager component depends on two AbCDServiceComponents, performance monitor and transaction management components. This can be visualised using a component dependency view, as shown in Figure 36. This diagram is an extension of the graph model generated using the Spring component framework and Eclipse development environment. The diagram shows a physical component dependency view of Bank example when using the Spring component framework.

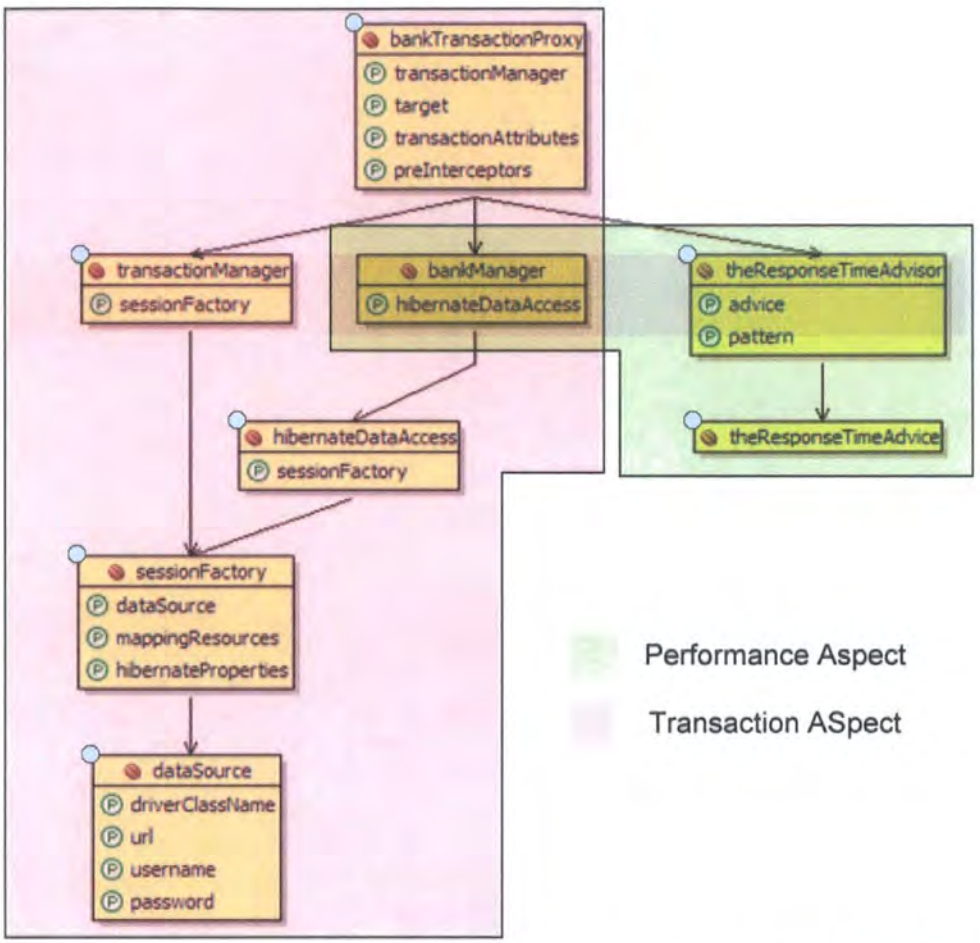


Figure 36 A simple component dependency view using coloured regions to represent aspects/services

As an overview, the view aims illustrate how functional components, that are applied with AbCDComponent stereotype, depend on non-functional and cross-cutting aspects in system design. The class diagram depicts two other important aspects of the component design as follows.

5.3.6.1 Component design transformation

It shows a mapping between a logical AbCDServiceComponent component and detailed physical components implemented using the Spring framework. A physical component is denoted with a circle on the top left corner if it is part of the implementation of an AbCDServiceComponent.

As shown in the diagram, using the Spring framework, the logical transaction manager AbCDServiceComponent is mapped to 6 physical components, where the bank component depends on them. It shows how the logical dependency between the BankComponent and the TransactionMananger is transformed to physical dependency between components. In other words, it shows how a logical dependency is transformed into a concrete dependency.

5.3.6.2 AOP view using colour regions

Figure 36 also shows an AOP view in component design. A coloured region highlights how a concern for an aspect is supported by a set of different components. A coloured region is formed by collecting physical components of the one or more AbCDServiceComponents that provide the same aspect.

The concept of colouring regions to express different aspects of the detailed design using UML graphical notation is derived from software visualisation [Stasko, Domingue et al., 1998]. The use of colours in UML modelling was first introduced in [Coad, Luca et al., 1999] with the notion of Archetype. Coad uses coloured model elements to layer different types of model elements. Different types of coloured model elements provide a visual constraint checking for the system design. The use of colours in this view has a different focus from what Coad has applied colours in UML. In this approach, colour regions are used as a visualisation support for understanding various aspects in system requirements in the design. Most importantly it presents non-functional and cross-cutting requirements based on the AbCD meta-model specification. Each aspect of the design can be assigned with a colour for graph generation.

In order to provide the component dependency view, a tool support is required as well as the mapping specification. The tool must support the following features.

- UML 2.0 profiling support for parsing AbCD profile.
- Mapping facility to allow the designer to define how abstract components are transformed to component framework specific models.
- Graph generator to present the model elements in coloured regions for analysis and design comprehension.

This research aims to provide an add-in support for an existing UML tool to support the designer with the construction of component dependency views using the AbCD profile.

5.3.6.3 An overview of the Component Dependency View

The main rationale for introducing this view is to extend the use of model driven approach with UML for the domain of component based software design. It is used as an application for the AbCD meta-model, to find out if the view can be generated from the class diagrams when applying the AbCD profile. The findings and extended discussion is made when presenting a case study in Chapter 7.

5.3.7 Technology dependency injection approach

Depending of the type of logical AbCD component, the developer can apply different attributes. As shown in Figure 37, these attributes can be used a filter different possible technologies or programming languages when implementing the component. In other words, attributes act as a meta-data for the specification which can be used when acquiring or implementation technology specific components.

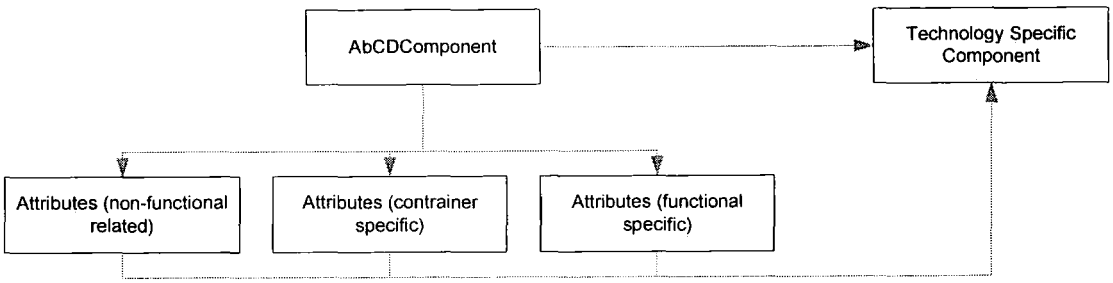


Figure 37 Applying attributes when implementing/acquiring technology specific components

For instance, the Bank manager component may be added with transaction specific attributes and the implementation may be only possible with particular technology or programming language. On the other hand, the designer can add attributes that may add technology specific dependency to the component.

5.4 Constructing the AbCD meta-model

The construction of the AbCD Meta-model started by defining the specification and the abstract model using the graphical notation. As part of the implementation, the AbCD meta-model is constructed using the Eclipse UML2 plug-in [Eclipse, 2005]. This is an existing open-source tool that is built as a plug-in for the Eclipse development environment tool to support the construction of UML profiles. The detailed description of the design and construction of the Meta-model is presented in Chapter 6 as part of the implementation and tool support for this research.

5.4.1 Expressing the model and tools support

One of the main objectives of this research is to allow the designer to express the meta-model using different syntaxes. This is based on the concept of separating the abstract syntax from concrete syntax that was introduced in OMG's MDA.

The abstract syntax for the meta-model can be expressed using different concrete syntaxes. One way to express the meta-model is using graphical notation as a concrete syntax. With the support of tools that provide UML graphical modelling and UML profiling, it is also possible to express the AbCD meta-model using graphical notation as a concrete syntax. On the other hand, Figure 38 shows a screenshot of a fragment of the AbCD profile, which represents the AbCD meta-model. The tree view shows the stereotypes that represent the meta-classes. Each stereotype has a variety of contextualised attributes as described in previous sections.

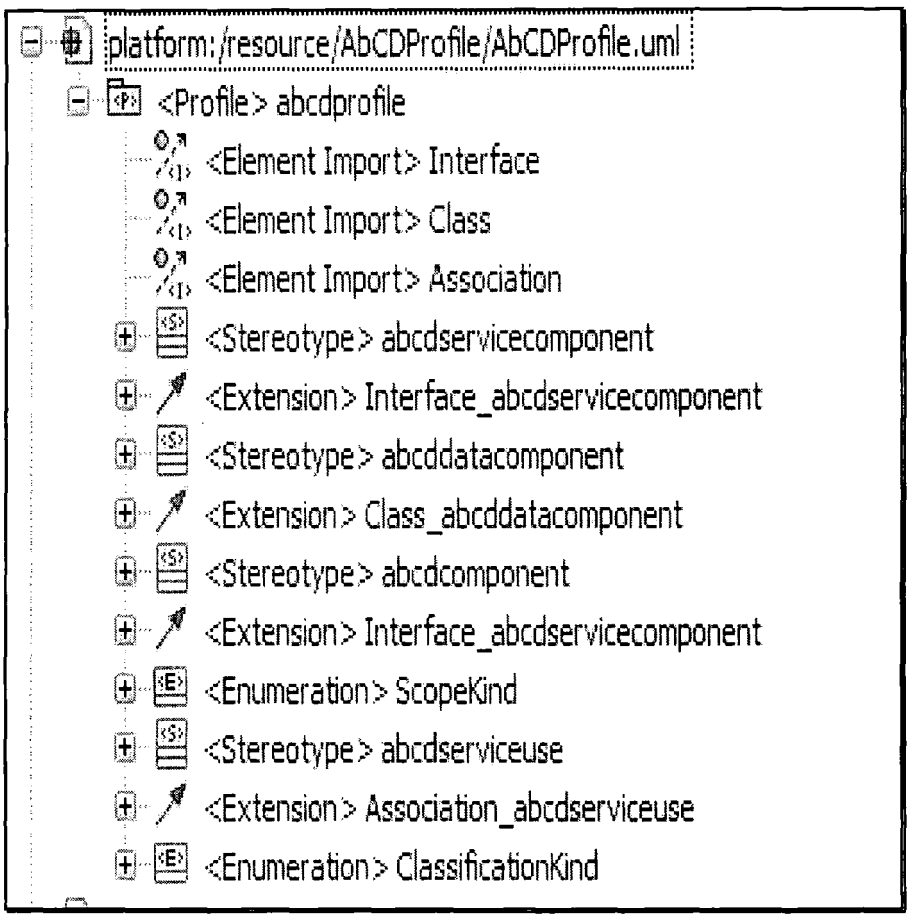


Figure 38 A tree view of the AbCD profile constructed using Eclipse UML plug-in

It is constructed using UML2 plug-in. The plug-in stores the profile as an XML file using the XMI specification. A fragment of the XML file is shown in Figure 39. It demonstrates the different representation of the model.

```
<packagedElement xmi:type="uml:Stereotype" xmi:id="_irKyQBlwEdum_ebbvynHnQ"
name="abcdservicecomponent">
    <ownedAttribute xmi:id="_DVdIEhvrEdutN_ER8sY21g" name="base_Interface"
association="_DVdIEBvrEdutN_ER8sY21g">
      <type xmi:type="uml:Class"
href="pathmap://UML_METAMODELS/UML.metamodel.uml#Interface"/>
    </ownedAttribute>
    <ownedAttribute xmi:id="_sggUsCYeEdutvPbNeN-Pcw" name="Name">
      <type xmi:type="uml:PrimitiveType"
href="pathmap://UML_LIBRARIES/UMLPrimitiveTypes.library.uml#String"/>
    </ownedAttribute>
    <ownedAttribute xmi:id="_vTzGQCYfEdutvPbNeN-Pcw" name="Scope"
type="_Gnqq0CYfEdutvPbNeN-Pcw"/>
  </packagedElement>
  <packagedElement xmi:type="uml:Extension" xmi:id="_DVdIEBvrEdutN_ER8sY21g"
name="Interface_abcdservicecomponent" memberEnd="_DVdIERvrEdutN_ER8sY21g
_DVdIEhvrEdutN_ER8sY21g">
    <ownedEnd xmi:type="uml:ExtensionEnd" xmi:id="_DVdIERvrEdutN_ER8sY21g"
name="extension_abcdservicecomponent" type="_irKyQBlwEdum_ebbvynHnQ"
aggregation="composite" association="_DVdIEBvrEdutN_ER8sY21g"/>
  </packagedElement>
  <packagedElement xmi:type="uml:Stereotype" xmi:id="_KeUoMBvrEdutN_ER8sY21g"
name="abcddatacomponent">
    <ownedAttribute xmi:id="_VKlWkhvrEdutN_ER8sY21g" name="base_Class"
association="_VKlWkBvrEdutN_ER8sY21g">
      <type xmi:type="uml:Class"
href="pathmap://UML_METAMODELS/UML.metamodel.uml#Class"/>
```

Figure 39 A fragment of the XMI file for the AbCD Profile

The UML2 plug-in tool allows the designer to construct UML models and apply the UML profile. Figure 40 shows a screenshot of the UML model constructed for the Bank example. It is applied with AbCD meta-model to construct component based design. The Bank UML model can be stored as an XML file based on the XMI specification.

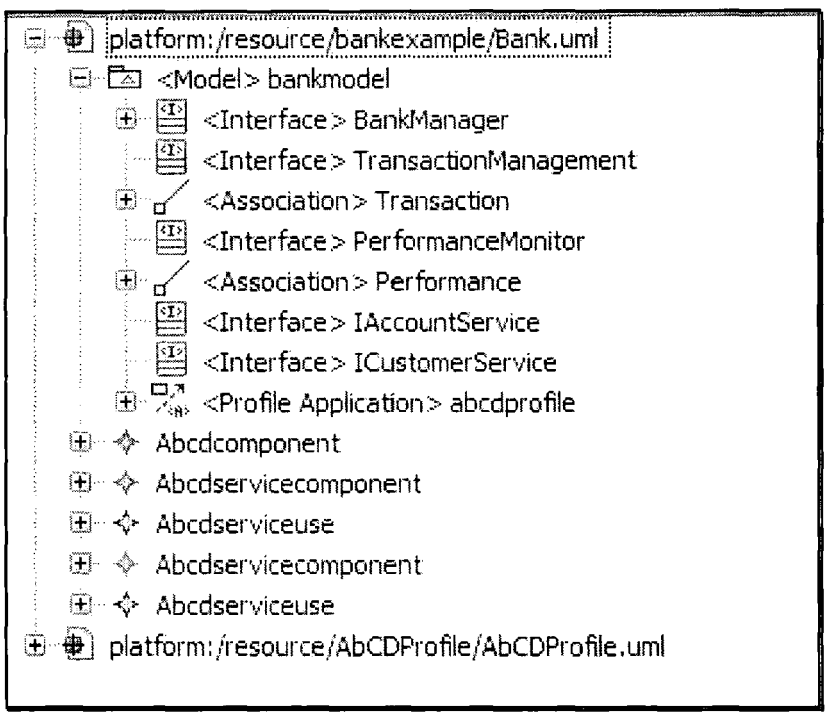


Figure 40 UML model for the Bank example

However the UML2 plug-in is a generic tool to support UML modelling tools with UML profiling support. This research aims to construct a new Eclipse plug-in, called '***AbCDTool***', to support the AbCD meta-model and component based design and development. Based on the XMI representation of the AbCD profile and UML model generated from the UML2 plug-in. The tool should provide the following features:

- It should generate a graphical model that depicts the *component dependency view*.
- It should also provide a parser that can perform analysis on the system design and report different non-functional aspects needed by various components.
- With the use of existing code generation tools, it support provide code/configuration generation support for a technology specific framework.

The AbCDTool will be used to evaluate the concept of applying the AbCD meta-model to model driven approach in component based design. The detailed evaluation is presented in 0.

5.5 An analysis of the Attribute based approach and the AbCD meta-model

When AbCD concepts were introduced in Chapter 4, six main principles of the approach were discussed in Section 4.2. This section summarises the work that has been done in the AbCD approach to accomplish the principles identified.

The first aim listed is to allow the developers *to identify the relationships between different business requirements based on component interaction using well defined interfaces*. To achieve the aim, the AbCD approach introduced AbCD meta-model which provides the construction of logical and abstract components. These mapping of classes from design to these components ensure that the component interaction is made using interface composition.

The second aim was to create *a framework that allows developers to define abstract level business components*. The AbCD meta-model proposes the AbCDComponent and AbCDServiceComponent meta-classes that explicitly extract functional, services and non-functional aspects of the business components at the design level.

The third aim was to *support developers when identifying reusable components*. As described in Section 5.3.7, the attributes defined in various components can be used as meta-data to filter components when implementing or acquiring technology specific and pre-built reusable components.

The fourth aim was to identify non-functional requirements that are overall system concerns to be resolved for all components, such as security, logging, and activation, etc. This aim can be achieved in two processes when using AbCD approach. Firstly, the designer can explicitly

define AbCDServiceComponents that represent non-functional and service requirements. Secondly, the AbCDServiceComponents can be used to construct component dependency view that depicts how functional components are depending on non-functional components.

The fifth aim was to allow the developers constructing components that are focused on aspects, contexts, abstraction and composition. The AbCD approach proposes the construction of components using Component Design Guidelines CDG. The CDG not only support the designers to identify aspects but also guide the designers to construct abstract components.

The sixth aim was to form a structure to build a reusable component model repository. Although the AbCD approach does not directly focuses on forming component model repositories, it is possible to construct searchable component repositories by using attributes as meta-data for components. Such repositories will provide better support for component reusability because the components in the repositories will be logical and components that are platform or technology independent.

Chapter 6 Implementation

6.1 Introduction

Chapter 4 highlights the need for a new model driven approach for the domain of component-based development. It also addressed the concept of Attribute-based Component Design AbCD and described ***Component Design Guidelines*** in Section 4.2.2. Chapter 5 supports the component design guidelines by detailing the AbCD meta-model specification and AbCD UML Profile. This chapter describes the implementation work carried out in this research. The work is to construct a set of tools, i.e. a ***tool suite***, to allow the developers to apply component design guidelines and realise ***context-based attribute injection*** introduced in Section 4.2. In other words, it allows the developer (i.e. the tool user) to define a AbCD UML Profile on the UML class diagrams and also perform analysis on component design.

Target	Description
System analysis and design modelling	Tools include features that allow developers to model the static structure and behaviour of the system for analysis and design. Depending on the UML version, commercial tools, such as [Together], provide a complete set of graphic modelling support.
Design Verification	With the use of constraint language, such as OCL, and cognitive functions, tools also provide design verification support for model driven development [ArgoUML].
Platform/Domain specific modelling	Tools also provide features to that allow developers to construct platform specific models for modern technologies such as .NET and J2EE. Furthermore, some tools also provide model transformation of platform independent models to platform specific models [ArcStyler].
Process modelling	Tools also integrate with popular development process, such as Rational Unified Process RUP. Accordingly, using the tool, development is tailored by the guidelines proposed by the process [Rational Rose].
Reverse and forward engineering using a target language	Tools provide facilities to reverse engineer the source code to graphic models. In addition, tools also provide facilities to generate platform independent code as well as platform specific code for a target language.

Table 10 Target areas of various MDA tools

There are a variety of research led, open source and commercial case tools to support model driven development. Table 10 summarises different target areas of model driven approach that various tools have been developed.

With the introduction of MDA with UML 2.0, many modelling tools have emerged in the literature.

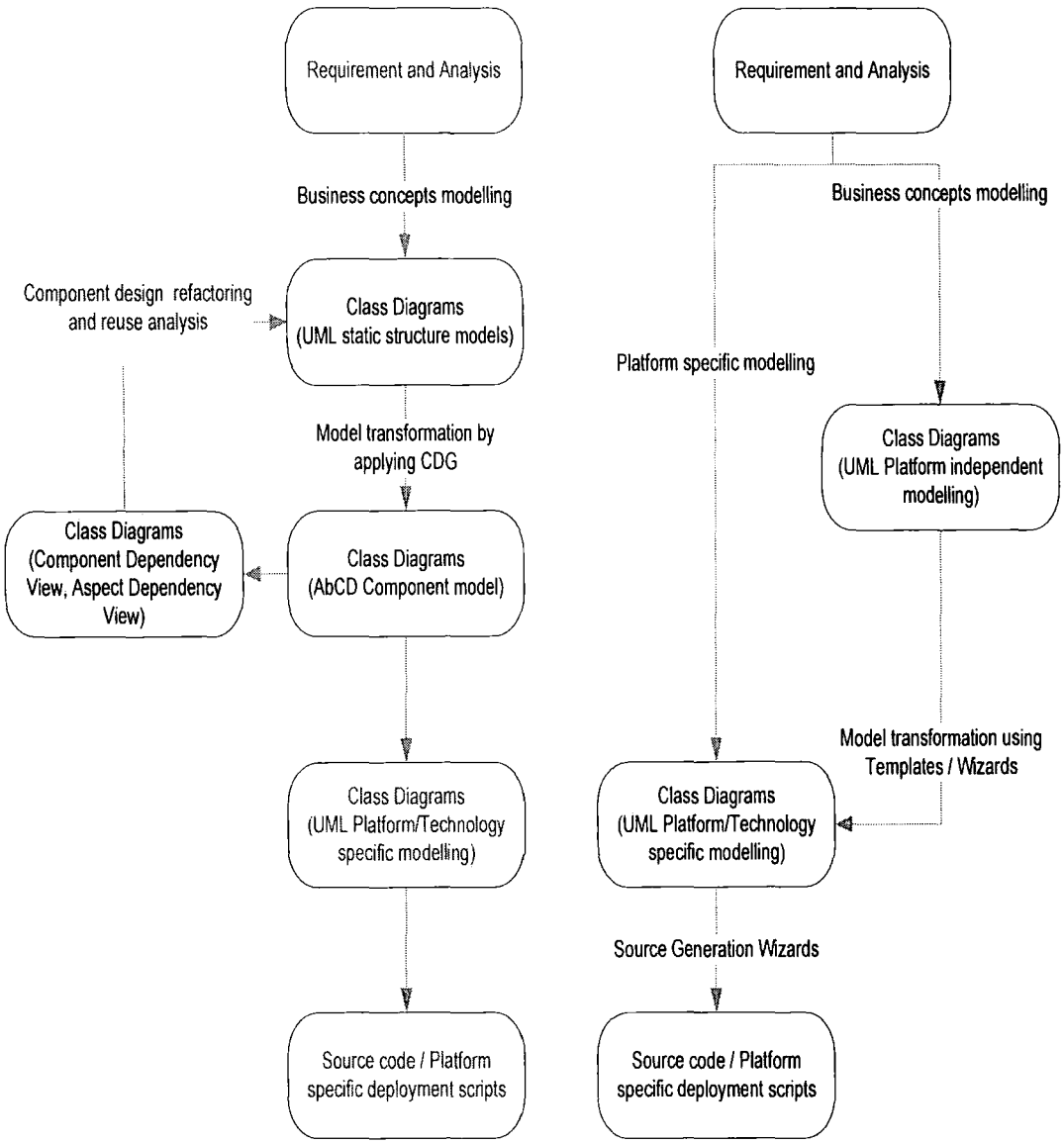


Figure 41 The tool suite targeting component modelling (left), other MDA tools targeting to bridge analysis to implementation

Tools, that support system design modelling, provide features that allow developers to construct the platform independent models and facilitate automatic transformation processes to the platform specific models. As an overview, Figure 41 (right) depicts the support provided by currently available modelling tools. The boxes represent the modelling artefacts that can be produced and directed lines show the modelling process. The aim is to reduce

development effort by providing code generation for different platforms. Additionally, these tools focus on bridging the analysis design to implementation.

On the other hand, the implementation work carried out here does not intend to compete with features provided by other commercial tools. The focus of the tool suite lies in system design modelling for the component-based system domain. In other words this implementation work ***focuses on component design abstraction rather than automation***.

As an outline, the main aims of the implementation is to build a set of tools, i.e. a tool suite, are as follows.

Aim 1: To allow developers to construct AbCD component models. In other words, the aim is to provide facilities for modelling of logical components that are abstract and platform neutral. This is achieved by an ***Eclipse Plug-in tool for AbCD UML Profile tool (Profiling tool)***.

Aim 2 : To provide component dependency and aspect dependency views using graphical modelling, allowing developers to perform component refactoring, dependency analysis and reuse analysis. This can be performed using the ***Eclipse Plug-in for Component Graph View and Aspect View tool (GraphView tool).***

Aim 3 : To implement a parser for the automation of model transformation processes. The aim is to allow the evaluation process of the AbCD approach by transforming the AbCD component model to two technology specific component frameworks. This can be processed by the ***Eclipse Plug-in for Spring Bean configurator and .Net C# generator tool (Generator Tool)***.

Figure 41 (left) shows the target focus area of the three tools implemented here. The highlighted boxes represent the target modelling artefacts to be produced and a basic workflow.

To summarise tool support, the Profiling tool takes the UML class diagram as an input to transform the system static structure diagram to an AbCD component model. The GraphView tool parses the AbCD component model to form a component dependency view for model analysis. The Generator tool can be used to transform the PIM to PSM using the lightweight Spring application framework or the heavy weight .NET framework. The rationale behind the implementation work is:-

- to evaluate the Attribute based Component Design approach by applying various case studies using the tool suite,
- and to assess the contribution of this research towards the component based development community.

6.2 Eclipse Plug-in tool for AbCD UML Profile (Profiling tool)

The eclipse plug-in tool for AbCD UML profile (or ***Profiling tool*** for short) is a plug-in for the Eclipse development environment [Eclipse, 2005]. It is designed to work with the existing Eclipse UML2 plug-in and Eclipse Modelling Framework (EMF) APIs. The Eclipse UML2 plug-in is an implementation of the UML 2.0 abstract syntax and conforms with the UML 2.0 specification, hence allows the developer to construct the static structure of the system as a set of UML 2.0 models. As shown in Figure 42, the central concept within the profile tool is to assist the developer to construct abstract and logical component model, i.e. the AbCD component model, from UML class models.

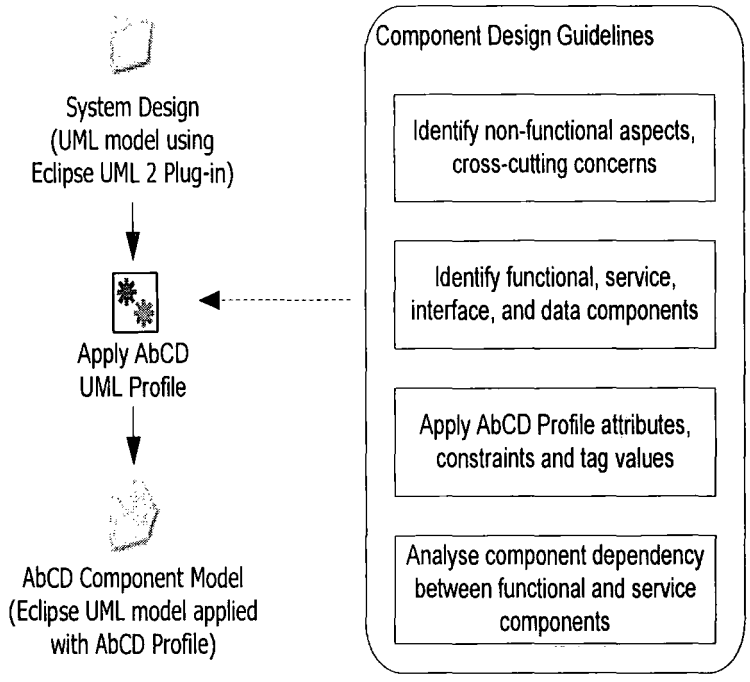


Figure 42 Applying AbCD approach using the Eclipse plug-in for AbCD profile

The UML 2 plug-in stores the model in XML format using OMG's XMI specification. However, the UML 2 plug-in does not include graphical modeller to construct and edit models as class diagrams. Therefore if a developer wants to use graphical modelling, there are also other Eclipse plug-in tools, such as [Jupe 2005; Omando 2006; Visualmodeller 2005], that provide graphical modelling support for the UML 2 plug-in.

6.2.1 The design and implementation of the Profiling tool

The first feature includes in the Profiling tool is the XMI import wizard which allows designers to import UML models constructed using UML 2 plug-in, as depicted in Figure 43.

The current implementation of the Profiling tool only allows the user to import XML models and not from other modelling tools. Once imported, the AbCD meta-model can be applied. The tool also includes event listeners that monitor the changes in the attributes of each applied components. This allows other extensions to register with the listener and analyse the changes in the attributes for automation such as code generation, search and filtering of components.

Figure 43 XMI import feature

In current implementation, this feature is used to notify events for AbCD Dependency view construction wizard.

6.2.2 Using the Profiling tool

To illustration the workflow using the profiling tool, the bank example, which was introduced in Chapter 4, is used. Amongst many artefacts produced as part of the system design, Figure 44 shows the four architecturally significant modules of the hypothetical bank example. They are: two types of bank clients, i.e. the web client (bank.web) and the GUI client (bank.client), core business model (bank.core) and the utility module used for providing facilities such as security, logging, etc. used by other modules (bank.common). The UML 2.0 model of the bank's core has also been developed using Eclipse UML 2 plug-in. Using the Omando graphical plug-in tool, the system design of the bank's core module was shown in Figure 20 in Chapter 4.

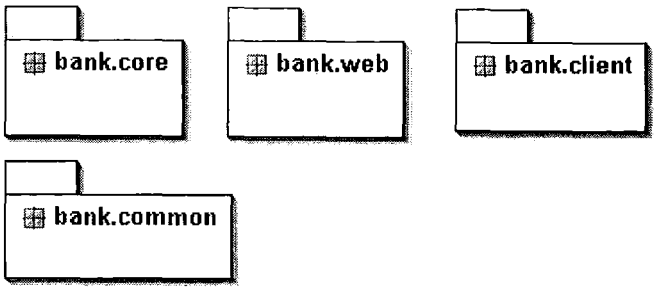


Figure 44 Architecturally significant modules from the bank example

To use the profiling tool, the system design, i.e. UML model constructed using the UML 2 plug-in, needs to be imported to the Profiling plug-in tool. Figure 45 shows two fragments of the tree view of the bank UML 2.0 model imported from the UML 2 plug-in.

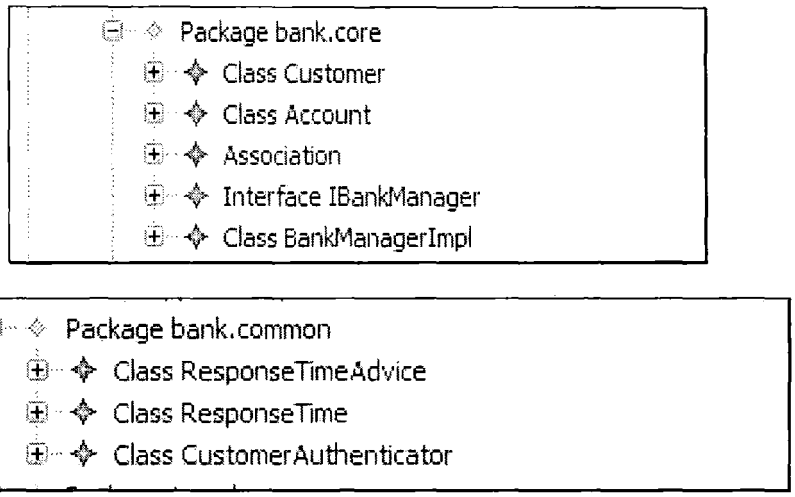


Figure 45 A screenshot of the fragment of UML 2 model from the bank example imported from UML 2 plug-in to Profile tool

Once imported, the model can be transformed into the AbCD component model by:-

- following the component design guidelines,
- and applying the AbCD UML profile.

As described in Chapter 3, the AbCD approach comprises component design guidelines which include **component identification**, **dependency identification** and **component construction** processes. Using these processes and the AbCD UML profile, the developer can use the profiling tool to:-

- Identify aspects, component service, and apply attributes
- Analyse component dependencies.

6.2.3 Identifying aspects and component service

This modelling phase is motivated by the concepts of Aspect Oriented Programming (AOP) and Component-based Software Engineering (CbSE). In the bank example, when UML 2.0 model of the core business concepts were imported, it is not clear how the model can be partitioned into logical components. When following the component design guidelines, the partitioning starts by identifying functional aspects, cross-cutting aspects and non-functional aspects of the system requirements.

The profiling tool supports this process by allowing the developer to construct a new model and add five types of logical components (i.e. stereotyped model elements) that are specified in the AbCD meta-model. They are as follows.

6.2.3.1 Implementation to support the AbCDComponent meta-class

A UML interface applied with the AbCDComponent stereotype defines an abstract component that performs ***functional aspects*** of the system. As shown in Figure 46, the tool allows the developer to apply attributes defined in the AbCDComponent meta-class.

Figure 46 Attributes of AbCDComponent meta-class

The attributes add the semantic information to the component such as information on its runtime environment, component activation type, etc. The detailed description of the AbCDComponent meta-class is described in Section 5.3 in Chapter 5.

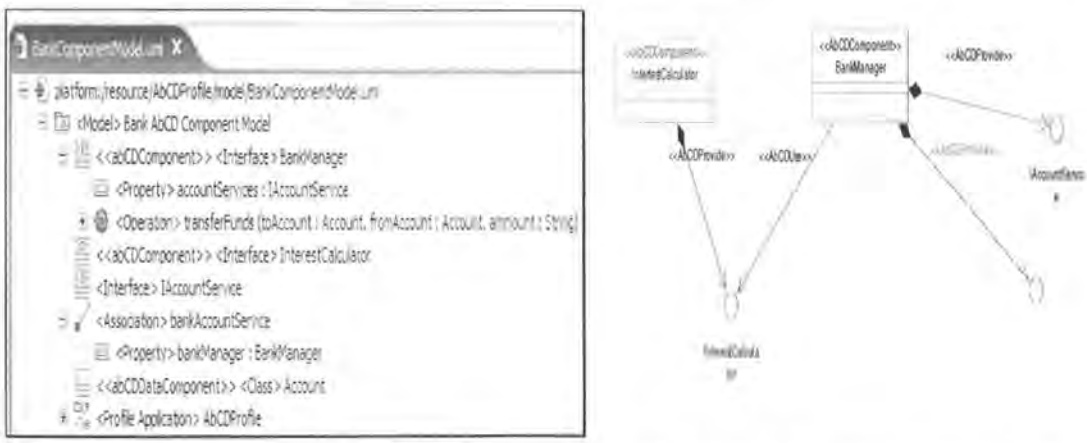


Figure 47 Applying the AbCD UML profile using UML 2 Plug-in (left Tree view, Graphical view right)

For the bank example, two AbCDComponents, i.e. BankManager and InterestCalculator, were identified. One of the important concepts of using this profiling tool is, as these two components are logical and abstract, each component can link to one or more physical classes and interfaces from the imported system design. Another important concept is that the profiling tool enforces the developer to use one of the main the CbSE concepts, i.e. Design by Contracts. This is achieved by limiting the component interaction to interfaces only, and by explicitly declaring the interactions using associations that are applied with AbCDUse, and AbCDProvide stereotypes. Detailed descriptions of these stereotypes are presented as part of AbCD Meta-model specification in Section 5.3.

Figure 47 (left) shows a screenshot of an initial version of the AbCD component model derived from the imported bank UML model. As the current implementation of the profiling tool does not include the graphical modelling support, the equivalent graphical representation of the component model constructed using the Rational Rose tool is shown in Figure 47 (right).

6.2.3.2 Implementation to support the AbCDServiceComponent meta-class

Another important feature supported by the profiling tool is allowing the developer to add logical components for non-functional and cross-cutting aspects. Based on the AbCDServiceComponent meta-class defined in the AbCD meta-model, an interface applied with the AbCDServiceComponent stereotype represents an aspect or a service required by AbCDComponents. Figure 48 depicts the AbCDServiceComponent stereotype and its attributes defined in the AbCD UML Profile.

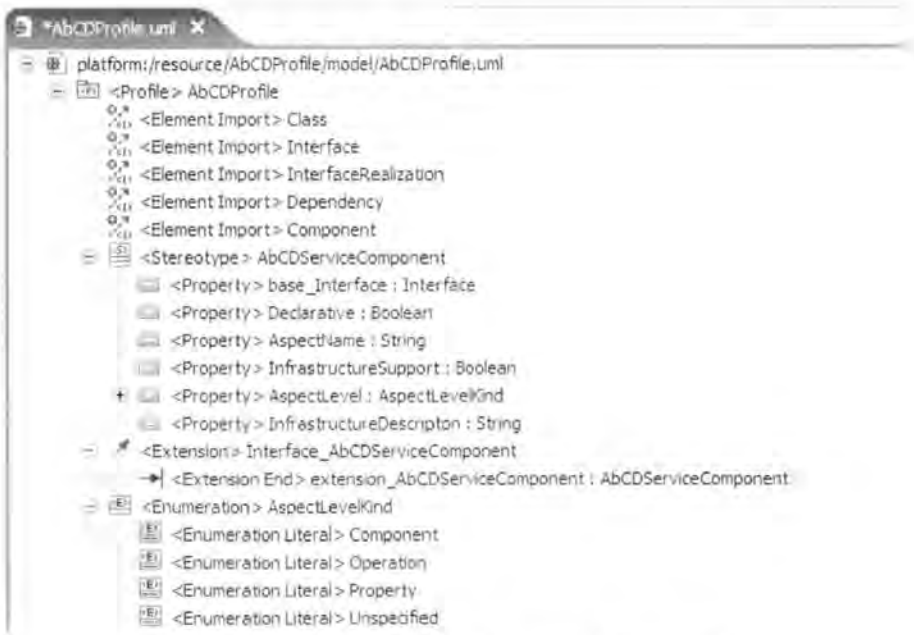


Figure 48 AbCDServiceComponent stereotype

Referring back to the bank example, there are five non-functional aspects required by the BankManager and InterestCalculator components. Accordingly, they can be identified as:-

1. Performance monitor AbCDServiceComponent
2. Transaction manager AbCDServiceComponent
3. Security AbCDServiceComponent
4. Tracing AbCDServiceComponent
5. Persistence manager AbCDServiceComponent

For instance, when one is setting a boolean attribute 'InfrastructureSupport' of the performance monitor to true, he or she is specifying that the component framework should include the feature to facilitate that aspect. Similarly, a declarative attribute can be used to specify the need for declarative approach to facilitate the aspect as opposed to a programmatic approach. The profile's attribute values are stored together with the model in the XMI file that represents the model, as shown in Figure 49.

```
<AbCDProfile:AbCDServiceComponent
xmi:id="_T7ujwEdOEdumV6ac2R_OeA"
base_Interface="_QghoMEdOEdumV6ac2R_OeA" Declarative="true"
AspectName="Performance Monitor" InfrastructureSupport="true"
AspectLevel="Operation" InfrastructureDescripton="The
implementation should use the AOP enabled component
framework."/>
```

Figure 49 AbCDServiceComponent stereotype data stored in the XMI file

This is an important feature included in the Profiling tool. This is because the information is used by the Graphview tool (see Section 6.2.4) to provide the analysis of the system design for technology feasibility. The detailed discussion on analysis of the system design is made in the next section.

Figure 50 Dependency between BankManager and Performance monitor components using AbCDServiceUse stereotype

Another two imperative attributes of the AbCDServiceComponent stereotype are 'AspectRequirement' and 'ImplementationClass'. Due to time constraints, the profiling tool does not implement this feature. However, if the profiling tool were to support the 'AspectRequirement' attribute, the developer can link the aspects defined in the requirement specification, such as UML use case[7], to each AbCDServiceComponent stereotyped element. Similarly, an 'ImplementationClass' attribute would allow the developer to define the technology specific implementation information for the aspect. This is one of the features that many MDA tools are trying to include, because it links the requirement analysis to design and implementation.

As described in the AbCDServiceComponent meta-class specification[8], when an AbCDComponent requires a particular service or aspect, the developer can add a dependency relationship with AbCDServiceUse stereotype to the AbCDServiceComponent. For instance, as shown in Figure 50 (left), using the profiling tool it is possible to define that the BankManager AbCDComponent logically depends on PerformanceMonitor AbCDServiceComponent. The same dependency is shown in Figure 50 (right) using the diagram. As it is a logical dependency, the technology specific classes that implement that PerformanceMonitor and BankManager components may have different dependency relationship mapping at the implementation level.

6.2.3.3 Implementation to support the AbCDInterfaceComponent meta-class

Another logical component defined in the AbCD meta-model is the AbCDInterfaceComponent meta-class. It defines a logical component that provides the user interface or interaction aspect with the system. The AbCDInterfaceComponent stereotype, which represents the AbCDInterfaceComponent meta-class is another stereotype supported in the profiling tool. Depending on the nature of the user interface, the developer can define a set of attributes for the target interface model, such as the Web interface, the Web services interface or the desktop GUI interface. When using the profiling tool to define values for attributes included in the AbCDInterfaceComponent, the three significant attributes are ProcessModel, Remoting, and PresentationStyle. It defines how the AbCDInterfaceComponents may interact with AbCDComponents.

Figure 51 Two AbCDInterfaceComponents defined in the bank example

cxvi

[7] i.e. assuming the developer would use the UML use case for modelling requirements using a UML tool that support UML use case modelling within the Eclipse environment

[8] See the full description of the AbCDServiceComponent meta-class specification in Section 5.3.2

Figure 52 Three important attributes of AbCDInterfaceComponent

For instance, a ProcessModel attribute defines a runtime requirement for the component, and the possible values are :- 'Inprocess', 'OutofProcess' and 'Remoting'. It also defines how loosely coupled the business component and interface components are. Referring back the bank example, Figure 51 shows the two AbCDInterfaceComponents defined to process and handle the Web interface. As they are logical components, implementation may be different depending to component framework, such as J2EE or .NET frameworks. Figure 52 shows the attributes selected for the BankCustomerController component.

6.2.3.4 Implementation to support the AbCDDataComponent Meta-class

The profiling tool includes special functions to process the model elements applied with the AbCDDataComponent stereotype. It represents a data object. An important attribute included in the AbCDDataComponent is the ComponentDataFormat attribute, and the possible values are JavaBean, XML, Binary, Custom, and Unspecified. This is framework-specific information about the data object. Unlike other components, the AbCDDataComponent meta-class is derived from UML 'Class' rather than 'Interface' because it signifies a more concrete representation. As described in the component design guidelines, the developer should explicitly apply data classes that are shared between components.

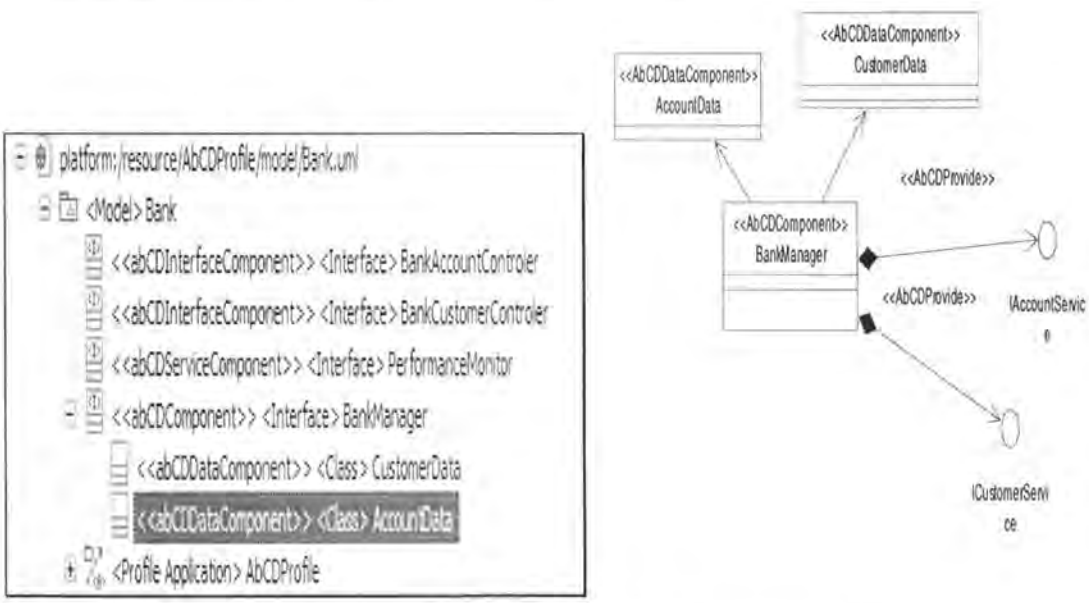


Figure 53 BankManager AbCDCOmponent

Figure 53 (left) shows the BankManager component publicly sharing two AbCDDataComponents using the profiling tool, i.e. AccountData and CustomerData. Figure 53 (right) presents the same information using the diagram. The AbCD meta-model does not address any data objects used within the implementation of the BankManager component. Current UML 2.0 specification defines the concept of using Ports to explicitly define operations and interfaces. The AbCD meta-model extends the same concept by explicitly defining the data objects exposed by the component.

6.2.3.5 Implementation to support the AbCDComponentAssembly Meta-class

In the AbCD Meta-model specification, AbCDComponentAssembly Meta-class can be used as a deployable component. Using the profiling tool, the developer can group a set of components that perform a particular aspect. It is similar to use of packages in UML modelling. However AbCDComponentAssembly contains the following attributes such as VersionInfo, Title, DeploymentDescription, which make the component self-contained and self-descriptive from the CbSE point of view.

6.2.4 Component dependency View (GraphView tool)

The section describes the implementation work that allows the designer to visualise the design model. It is implemented as a graphview plug-in to Eclipse IDE and it requires Profiling tool and UML2. Furthermore, the functionality of the tool heavily replies on Graphical Editing Framework (GEF) package from Eclipse project [Eclipse, 2005]. The GraphView uses GEF to display AbCDComponents as UML class diagrams and provide two main views as follows.

- Reusability View: The tool uses event register functionary from Profiling tool to monitor changes in attributes of various attributes. It then parses the attributes for possible reusability of existing components for a particular AbCD component. The tool analyses two types of AbCD components. Firstly, it analysis AbCDServiceComponents for possible matching of a component framework that supports the attributes specified in the AbCDServiceComponent. Currently the database has .NET, J2EE and Spring framework information. It is possible to add additional framework information to the database if needed. Secondly, it analysis AbCDDataComponents for possible matching of existing persistence models provided by different frameworks.
- Dependency View : The tool also includes dependency view. It provides coloring regions that presenting AbCDServiceComponents. It depicts how AbCDcomponents depend on AbCDServiceComponents.

6.2.5 Code generation Process (Generator tool)

As part of the implementation for AbCD approach, a basic Generator tool is also constructed. It is constructed with a basic functionality to illustrate that AbCD approach can be used to construct ground-trip development. The tool aimed to provide code generation support for .NET framework based components using C# as will as Spring framework based components using Java. However the language parsers are not implemented yet. This is because the code generation is not the main focus of this research and there are many existing code generation tools available in the literature.

6.3 Summary

This chapter presented the implemented work done to support the AbCD approach and the meta-model constructed using UML. It showed how different Eclipse plug-ins are constructed that allow designers to construct AbCD component models at logical level. However the implementation is heavily replies on the Eclipse IDE and other eclipse plugins. Therefore the main limitation is the user is forced to construct models using Eclipse IDE. The success of the AbCD approach and the usefulness of the models constructed relied not only on how the designer apply the AbCD approach but also the support of the tools provided to implement the model. Although basic implementation is completed there are many improvements to be made as a further research.

Chapter 7 Case Studies

7.1 Introduction

This chapter describes case study work carried out to support the evaluation process for this research. So far in the thesis, only the banking example, which was introduced in Chapter 3, is used to illustrate the concepts. This work includes the study of two software development cases covering different scenarios of the component-based software development method. Each case study is focused on different aspects of component development using the model driven approach. In other words, each case study comprises a set of characteristics that aim to measure the contribution of the concepts introduced in the Attribute based Component Design approach. The following sections describe each the case study in detail and illustrate the development process using the tool suite presented in Chapter 6. This chapter can be regarded as a reflective report on the two main case studies carried out part of the PhD research. It is also an evaluation work to investigate the AbCD approach using two different domains.

7.2 Case Study 1: myanmarshop.com ecommerce website

Before describing the case study in detail, it is necessary to outline the main reasons for choosing to use this case study. This case study is used to assess the Attribute based Component Design (AbCD) approach in the following ways:

- The case study is to examine if the AbCD approach can assist ***the construction of streamline traceability between requirements specification and design of the components***.
- The ABCD design concept was intended to be applied to large-scale enterprise applications that require services in different contexts. The evaluation will be made on the component specification model constructed using AbCD concepts to assess if ***the model can help developers choose appropriate component based technology***.

7.2.1 Background

A company called 'myanmarshop.com' wants to develop an e-commerce application as an online business for retailing and wholesaling imported foods and other products from south-east Asia. The application will be based on the existing business process, practices and infrastructure. As a rapidly expending organisation, the main requirement that is proposed to the development team is that the design of the application should be agile. In other words the design should be easily extendable with exchangeable components that are **not specific to a technology and implementation neutral**. This is because both tight and loose coupling is

needed with various types of business partner's applications to share information and integrate business processes.

Figure 54 show a high level overview of system requirements for its three business sites.

The organisation has four warehouses where the products are stored. It wants to develop a global store front with localisation aspects integrated to the system. Using Web Services technology, it also wants to integrate with over 30 supplier companies to automate the process of streamlining their upstream procurement activities. The system deployed at each of the business sites should be integrated to form a global e-business system. With the use of RPC calls, components deployed on each site will be tightly coupled using the private virtual network over the internet. It will also reflect the organisation's structure and provide centralised management.

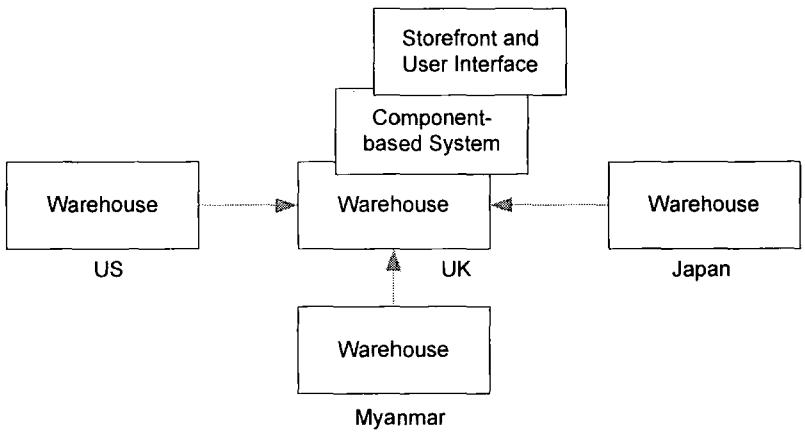


Figure 54 myanmarshop.com business sites

Presently, there are various state of the art enterprise-level component-based implementation technologies available such as COM+/.NET, J2EE, Web services and CORBA technologies. However after the initial analysis the following requirements can be outlined:-

- The design of the components within the system cannot be based on a model that is specific to a technology because of the heterogeneous nature of reusable COTS components that should be applied to enable cost-effective and rapid development.
- A generic specification, which includes functional aspects such as correctness and functionality compliance as well as non-functional aspects such as performance, instance management and security, for each component is needed.

Accordingly a component specification model is needed to allow the developers to perform development as well as component acquisition processes. The construction of such a component specification model will provide the developers with an additional abstract layer of development and the architecture of system as shown in Figure 55.

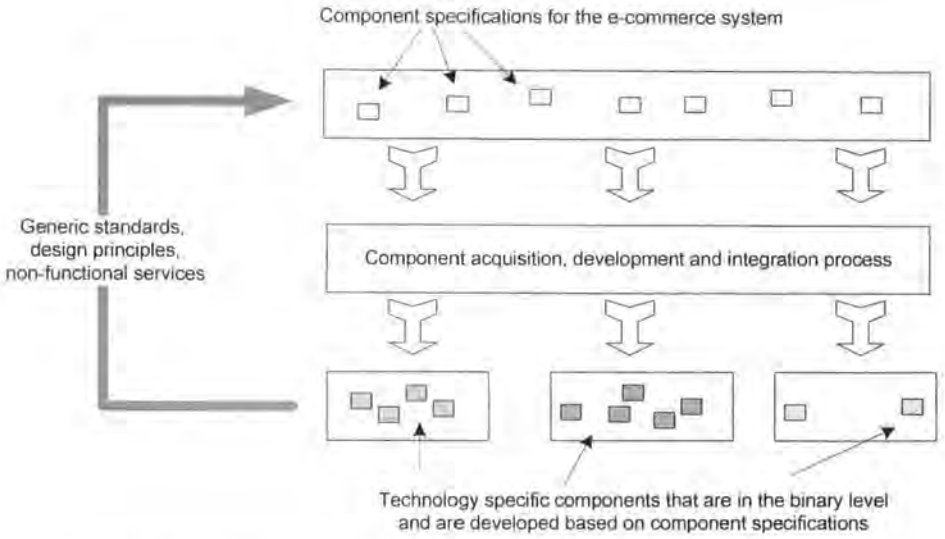


Figure 55 Component specification and implementation mapping

The availability of such a component specification will also produce the architecture for the system which shows various abstract views. The following section describes how ABCD is applied to both design and development of the system.

7.2.2 Designing the myanmarshop.com eCommerce system

This section describes how the design of the e-commerce system is evolved using the component based approach. However it does not detail all parts of the design and only highlights the architecturally significant modules. In this development, the developers agree to follow the Rational Unified Process (RUP) as a main discipline and apply the AbCD design principles as a supplement to the main workflow. This configuration of workflow also provides an opportunity to assess whether the AbCD approach can contribute to improve the design, and hence implementation phase. It can also be regarded as a non-invasive process because designers were able the transformation of their class modelling to AbCD component models in every iteration of their main design workflow.

Figure 56 shows an overview of the workflows. The analysis model is a UML model constructed using the Rational Rose UML modelling tool. Based on RUP, the designing of the system is an iterative process that adapts to the changing requirements.

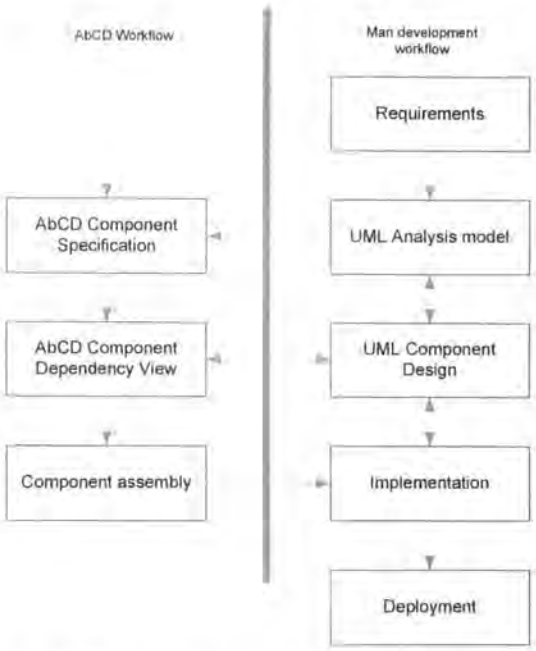


Figure 56 The workflows of the development process

The analysis model is composed of over 40 classes that represent the functionality of the core business. The classes are grouped into 16 packages based on their functionality. There are also over 30 data objects identified in the model, ranging from products, manufacturers, to localisation objects. These are platform independent UML classes constructed using Rational Rose UML model.

7.2.2.1 Deriving the AbCD component specification

As a separate model process, an AbCD component specification is formed. The aim is to derive the AbCD component specification using the UML analysis model as an input. The specification consists of a new set of UML class models applied with the AbCD UML profile. During the construction of the specification, the following concepts were applied based on the Component Design Guidelines (CDG) presented in Chapter 4.

7.2.2.1.1 Identification of components

This process starts by refactoring the classes. In the first iteration, the classes were grouped into three types of logical components, namely: interface, business and data components. They can also be regarded as components that represent the three tier architecture. The business components are further refined based on their *aspects* and *responsibility*. The business components were then divided into core functional components, cross-cutting functional components and service components.

One of the main concepts applied in the component identification process is *the partitioning of the system using interface composition*. In other words, component interaction is made using their façade interfaces, which ensures ***that the dependency relationship is formed using interfaces as contracts rather than inheritance***. The difference between interface composition and inheritance dependency is presented in the literature survey. The following section describes these concepts by an example.

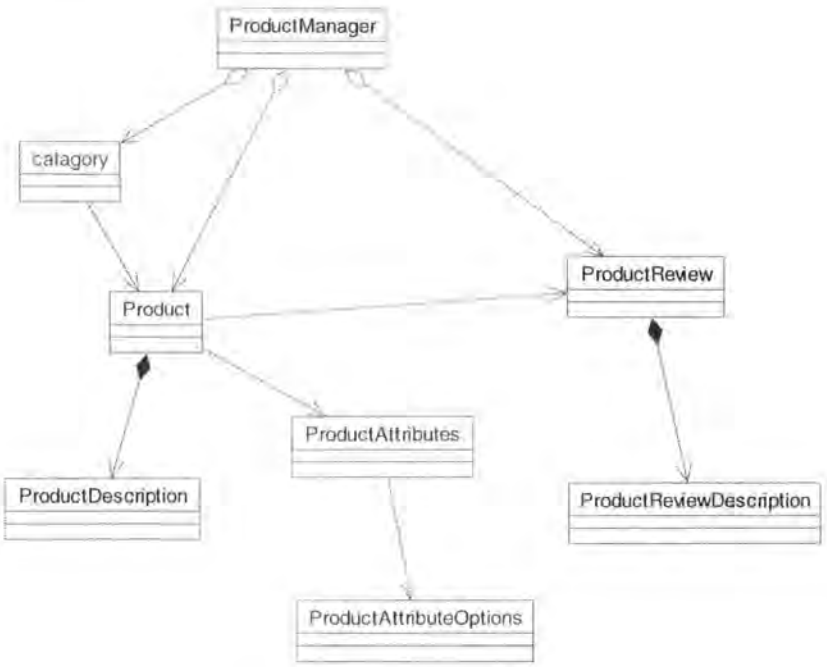


Figure 57 myanmarshop.com's products model

Figure 57 shows a snapshot of a UML diagram that is part of the UML analysis model. It includes the main classes and their relationships that form a core functional aspect, 'product management'. For simplicity, all operations and properties have been removed from the classes. Figure 58 shows another snapshot of the design that depicts how the design was changed when classes providing the localisation and language support aspect are added to the classes to the design model. The localisation and language support is a cross-cutting aspect, covering all other aspects of the e-commerce system, such as customers, product suppliers, user interface and currency management.

Although it is a valid model from object oriented design point of view, the design needs to be re-factored based on CDG for the following reasons:-

- The design of the cross-cutting aspect is embedded within other core functional aspects. Accordingly different aspects of the design are tangled and will be difficult to maintain.
- The current design does not apply interface composition approach across different aspects. CDG states that all dependencies should take place in the form of interface composition.

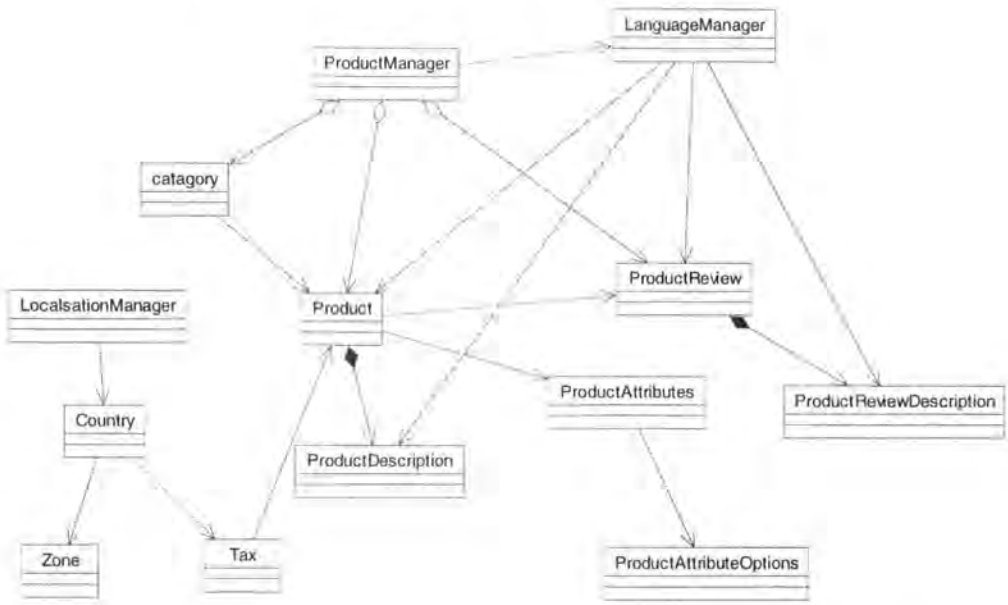


Figure 58 Introducing a multi-language support as a cross-cutting concern

Figure 59 shows another modification made by the designers to add a persistence aspect to all data classes. Without describing the implementation details, the designers add the 'Serialiser' class which all data classes must inherent from to achieve persistence. This produces a more tangled design, which can lead to reusability problems. The construction of the AbCD component specification aims to separate the concerns by forming an abstraction model.

During the component identification process, a new set of logical components are constructed. Based on the AbCD meta-model, four types of components are formed, namely AbCDComponent, AbCDDataComponent, AbCDServiceComponent and AbCDInterfaceComponents.

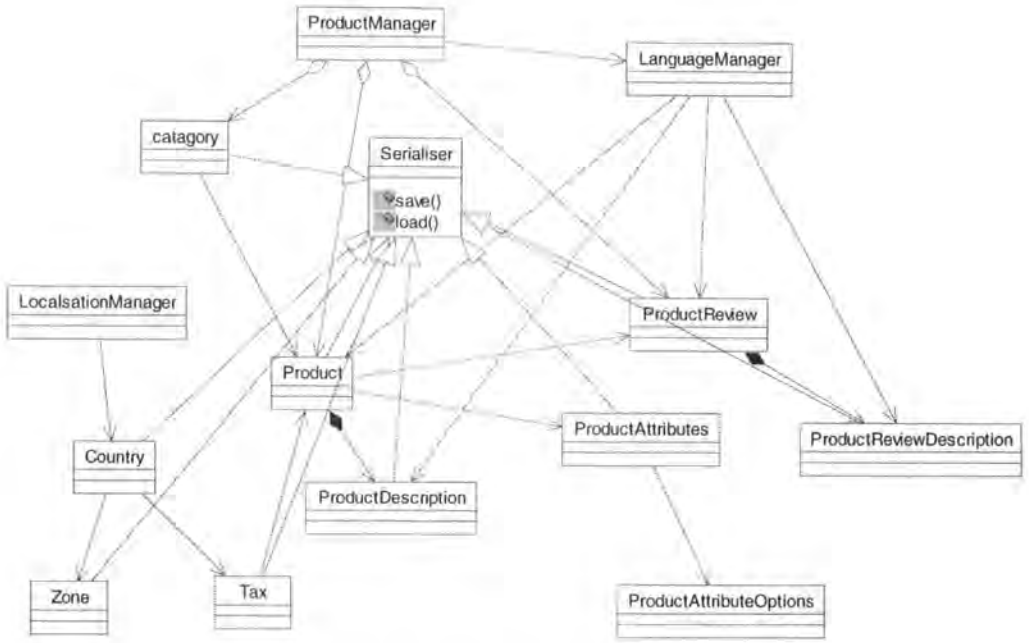


Figure 59 Introducing Serialiser class for persistence service

Each logical component is mapped to a set of classes for a particular aspect. The mapping is supported by the profiling tool. Figure 60 shows three new logical components as part of the AbCD component specification. This model provides the designers with clear separation of concerns and an abstract view.

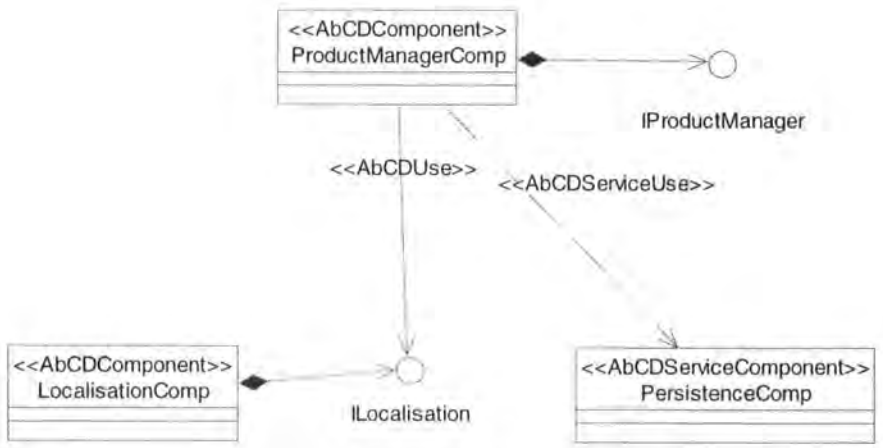


Figure 60 A snapshot of the AbCD Specification model

In this diagram, one of the important aspects to be observed is the dependency relationship between the PersistenceComp component and the ProductManagerComp component. Even though it is modelled with a simple UML dependency relationship, the AbCDServiceUse stereotype adds additional semantic information to the relationship. Any relation defined

using the AbCDServiceUse stereotype forms a contract. In this contract, the AbCDComponent must meet the terms defined in the AbCDServiceComponent in order to use the service. Based the AbCDServiceComponent meta-class, any model element applied with the AbCDServiceComponent stereotype has three parts of behavioural attributes.

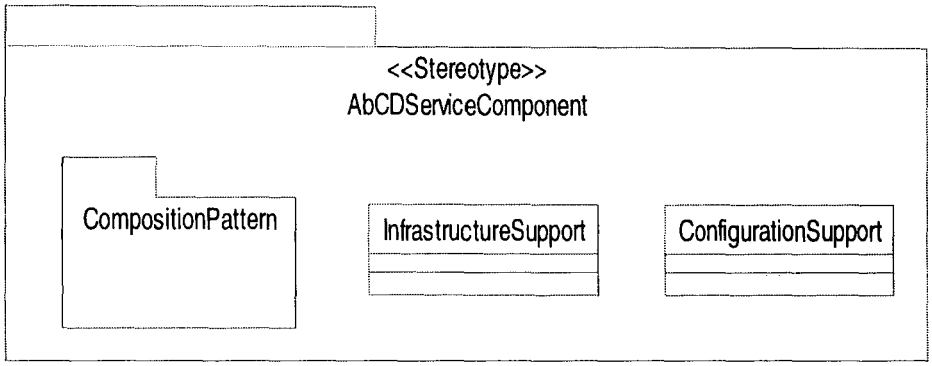


Figure 61 AbCDServiceComponent specification

As show in Figure 61, the attributes of the AbcDServiceComponent are as followis :-

Infrastructure Support : This is an attribute set by the designer that describes how a service can be supported by the framework it will be built on. In other words, if the service should be provided by the component framework, the design has to describe how an existing component provided by the component framework will provide the service. Whether or not the service is provided by the framework, the designer also has to define the detailed binding structure between the AbCDComponent and AbCDServiceComponent using the 'Composition pattern'.

Composition Pattern: Each AbCDServiceComponent includes a composition pattern. It describes how any class that requires the service may bind to it using a particular pattern. As described in the literature, if the service is provided by the framework, the designer is required to have in death knowledge on the component framework. Due to the diverse nature of each of the service, the composition pattern can be varied. Therefore, the CDG does not impose any particular model. One way to describe composition pattern is with the use of UML template bindings as described in [Clarke, 2003].

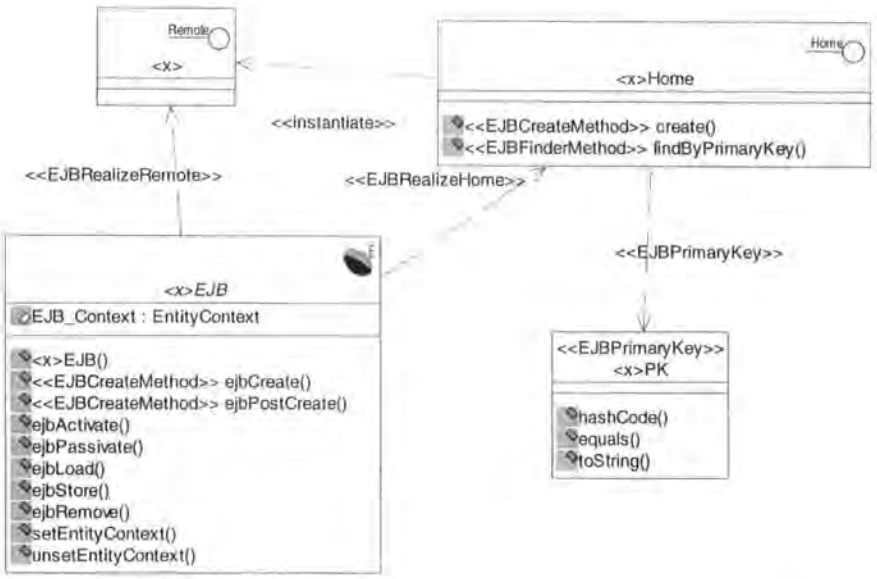


Figure 62 Compostion pattern for persistence model using J2EE

For this case study, the designers have decided that the persistence service should be provided by the component framework. Accordingly, if the components were to be implemented in Java, the two candidate component frameworks are J2EE and the Spring application framework. If J2EE were to apply, any class requiring persistence service has to transform into an Entity bean, as shown in Figure 62. In the figure, <x> represents a template parameter which should be replace by the actual class. The main advantage of the using attributes based approach is that such composition patterns can be added as attributes for a target technology. Therefore designers will be able to see how is design is altered when transforming platform independent design to platform specific design.

However, the designers have decided use the Spring framework to provide the persistence service. Since it allows transparent data access layer, it does not invade the core functional service.

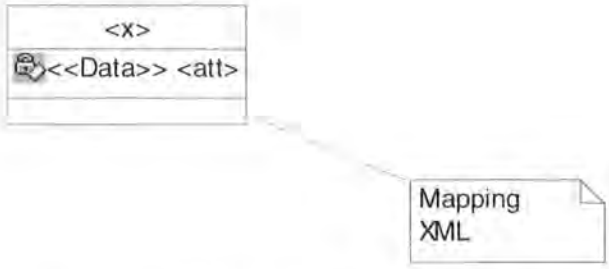


Figure 63 Composition pattern using Spring framework

As shown in Figure 63, the composition pattern is simple when using the Spring framework. <x> represents a template parameter which should be replaced by an actual data object and

any attributes applied with <<Data>> stereotype should be included in the XML object relational mapping file for persistence. However, to provide non-invasive approach, the Spring framework uses one or more external configuration files, in this case, XML mapping files are required to achieve persistence for data objects. In a similar way to the Spring framework, other component frameworks may require external configuration files to support various services such as tracing and caching data. Accordingly, the AbCDServiceComponent meta-class also includes another attribute called 'Configuration support'.

Configuration Support : This is another attribute defined in the AbCDServiceComponent meta-class is the attribute to provide configuration information. It contains information about configuration settings needed by the component framework in order to provide a particular service. Therefore, if the service provided by AbCDServiceComponent is implemented by the component framework, the designer has to fill in the configuration support attribute.

Figure 60 also shows the dependency relationship between LocalisationComp and ProductManagerComp. In this e-commerce application, the ***localisation aspect*** is important because the application will be deployed in 4 countries. The localisation aspect cross cuts all functional aspects, such as product management, order management, as well as all parts of the user interface aspects. Although the initial analysis model implicitly includes this aspect into classes that perform other functional aspects, the aspect was an afterthought and modelled in an ad-hoc fashion. The explicit declaration of this cross-cutting aspect as a separate logical component adds the following semantic information to the design and architecture:

- Designers are more aware of the impact on the design when the model needs to be refactored to accommodate the localisation aspect in data model, business functional model as well as interface model.
- Designers have to define a new dependency relationship between LocalisationComp and other functional and user interface components to achieve low coupling. Accordingly, adding a new locale to the application can be achieved without altering other modules.

Designers now have to be aware of the support provided by the component framework and the programming language that the application will be built on. In this case, the Java language and J2EE framework consists of standards and readily available resource managers that support localisation.

7.2.2.2 Summary of constructing the AbCD Specification model

To summarise, every non-functional aspect or service identified in the analysis model in the case study was applied as AbCDSeriveComponent stereotyped logical components. The

following AbCDServiceComponents were identified from the analysis model of the e-commerce system.

- Caching
- Tracing
- Transaction
- Security

For each component, the designer defines component composition attributes, framework support attributes, and configuration attributes. These service components, filled with attributes values, form the building block of the component architecture. One of the main advantages of explicitly identifying service components at early stage of the design is that the designer can rely on the attributes identified in the service components to form a contract with other functional components without the knowledge of their internal implementation details.

The component identification process continues with the classification of cross-cutting aspects from functional aspects. The other cross-cutting aspects that were explicitly identified are as follows:

- Language localisation
- Currency
- Tax
- Report Manager

The designers were challenged with the following issues in the design construction process.

Applying AOP techniques : The AbCD approach defined in this research does not propose a new AOP technique or approach that designer should apply for each cross-cutting aspect, because the components identified are abstract and the implementation may be different on the nature of the aspect. However, the explicit identification of cross-cutting and service components encourages the designers to refactor the design into components at early stage of the development cycle. These components may also take advantage of the services and AOP features provided by the component frameworks. However by applying attributes such as composition pattern and describing infrastructure support, the components encapsulate the implementation details.

Balancing the use of interface composition and contracts : For each component, designers were able to apply core CbSE features such as versioning, interface based composition, and interfaces as contracts.

For instance, consider the ManufacturerManagerComp uses the IReportinterface provided by the ReportManagerComp. As ReportManagerComp is an AbCDServiceComponent, it represents a logical component that provides the reporting function. By defining this component, designers were able to define composition patterns for other components to use.

7.2.2.2.1 Component construction

In this case study, the construction of the AbCD specification model from the UML specification model was proven to be difficult because of the following reasons:-

Tool integration: The main tool used in this modelling process was the Rational Rose UML modelling tool. Due to the lack of tools that can import Rose UML models to eclipse UML2 model, the UML analysis model has be duplicated using the eclipse UML2 plugin. It was an unnecessary step in the development. However once the XMI file generated from the UML2 plugin was imported to the AbCD profiling plug-in, a new UML model, that represents the Component specification, was able to be constructed using the profiling tool. As described previously, the components identified during the component identification phase were constructed by applying the AbCD meta-model and by using the Component Design Guidelines.

Attribute injection: With the help of the profilng tool, designers were able to add various attributes to all model elements applied with the AbCD UML Profile. Using the AbCD analysis tool, the designers were able to input the component specification for analysis of the components to visualise how functional, non-funcational and other service based components are related. However, the designers were unable to clearly visualise the component model and how each logical component can be mapped to the design model constructed in the main workflow using the Rational Rose tool.

7.3 Case Study 2 : Rapid Prototyping machine controller

When comparing with the e-commerce application presented in case study 1, this case study is vastly different. This is because of the different nature of the application to be designed and the focus lies in different evaluation criteria on the AbCD method. While the e-commerce system for case study 1 focuses on designing business components for functional and non-functional requirements this case study focuses on the following concerns:

- Extension and adaptation of existing components.
- Various existing ready-made components and services are needed to be reused for swift development.
- Extensive flexibility and expendability in design, based on framework support.

Therefore the focus on this case study is reusability of the components. The rationale behind choosing this case study is to assess if the AbCD approach can be used to assist the design using existing components and re-factor them based on core CbSE principles. In other words, while case study 1 centered around *abstraction*, this case study is to evaluate if the AbCD approach improve *reusability* of components. In this case, the ABCD is intended to help the developers to simplify the development process and to integrate components more efficiently.

7.3.1 Case study background

The manufacturing sector from Cardiff University wants to quickly develop software by integrating existing tools. They would like to develop a rapid prototyping machine controller that allows the engineer to correctly configure the rapid prototyping machine.

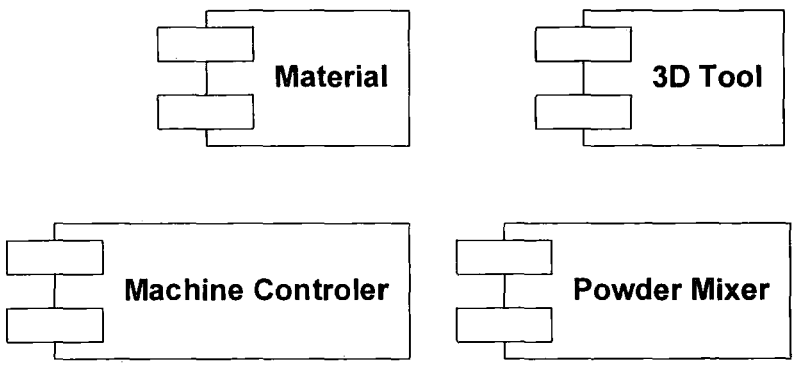


Figure 64 Four functional components of the tool

As depicted in Figure 64, they want to integrate a 3D modelling component, a powder mixing component, a material analyser component and the rapid prototyping machine controller

components. These are existing pieces of software from different PhD projects. Currently, there are number existing the applications for each component by various projects. Therefore the main development task here is component acquisition and integration rather than implementation.

The functional requirements of each component are not important in this case study. The main non-functional requirement is that the application has to be a distributed system where the engineer should be able to monitor from anywhere using a client application.

7.3.2 Designing the Rapid Prototyping (RP) tool

The components identified in the case study background are generic components which can be regarded as a Commercial Off-the Shelf (COTS) components. Initially, to form an application, developers were trying to construct adaptor code that bridges all four components based on their API. Creating such integration modules without a component framework created numerous problems. Theses can be outlined as follows :-

- Semantic integration problems: Different components use different meanings in their APIs.
- Architecture Mismatch: The components have their own architecture style. This has led to having their own model of interaction.
- Steep Learning Curves: different programming languages have their own structure and object management solutions.

This can be the result of not having a common framework. One of the core principles of CbSE and from this research point of view, all components must have common component framework. However in the case study it does not fit this requirement.

The first step of designing the RP tool is to define logical component based on existing components to form a common interaction model. From avoid confusion in this discussion; these COTS components will now be called 'modules'.

The construction of logical components should provide an indication on how these modules can be regard as atomic (i.e. how these components hide their functional and non-functional requirements). The AbCD approach will be used to construct the logical component model at the specification level to depict a higher component dependency view. This should provide a common component framework where each module must meet the specifications and attributes identified in logical AbCD components.

7.3.2.1 Deriving the AbCD component model

Figure 65 shows the process when applying the AbCD component model. It shows an iteration cycle for a component acquisition process. It is based on Component Design Guidelines defined in Section 4.2.2. The process is different from traditional software development because it focuses on component acquisition and integration of components rather than implementation.

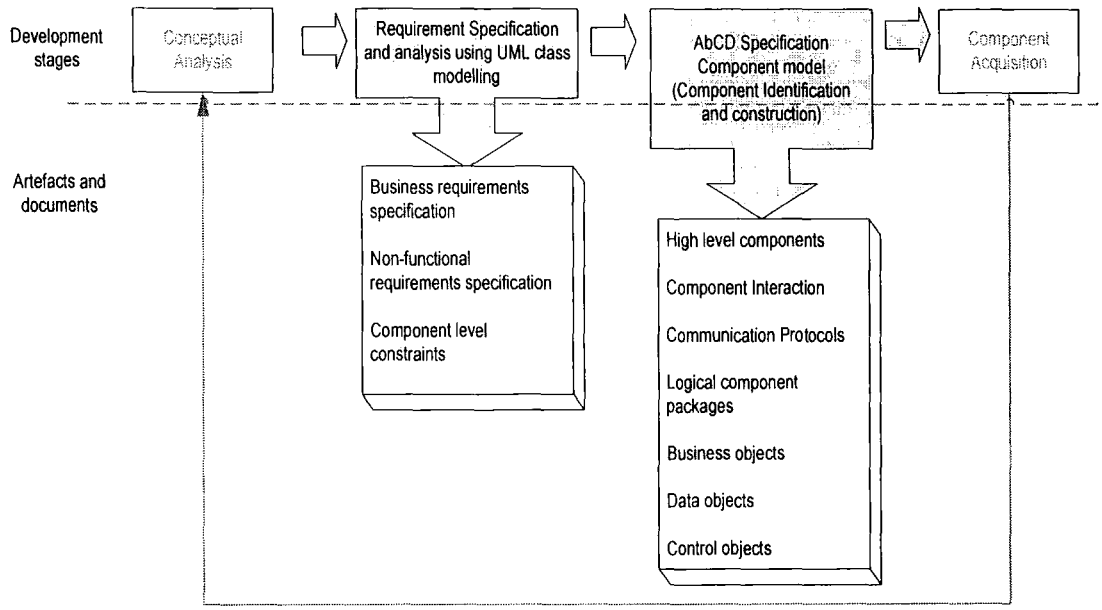


Figure 65 The modelling process for the RP tool

When following CDG, the first question raised by the AbCD approach is :-

> "Does each module conform to the common component framework of the RP tool?"

To find out, a new UML class model has to be derived for each module using a reverse engineering tool. This is necessary for understanding the functional aspects of the modules. However the reverse engineering tool has produced a large model of UML with complex interaction and dependency relationships, which does not help the designers. The first stage of the Component Design Guidelines is the identification of logical components.

7.3.2.1.1 Component Identification

From the requirement analysis, developers have derived logical AbCDComponents to form the architecture. Figure 66 shows a simplified version of the AbCD component model. To avoid complexity, all functional interaction and dependency relationships have been removed. Furthermore, data objects that are used to share information have also been removed.

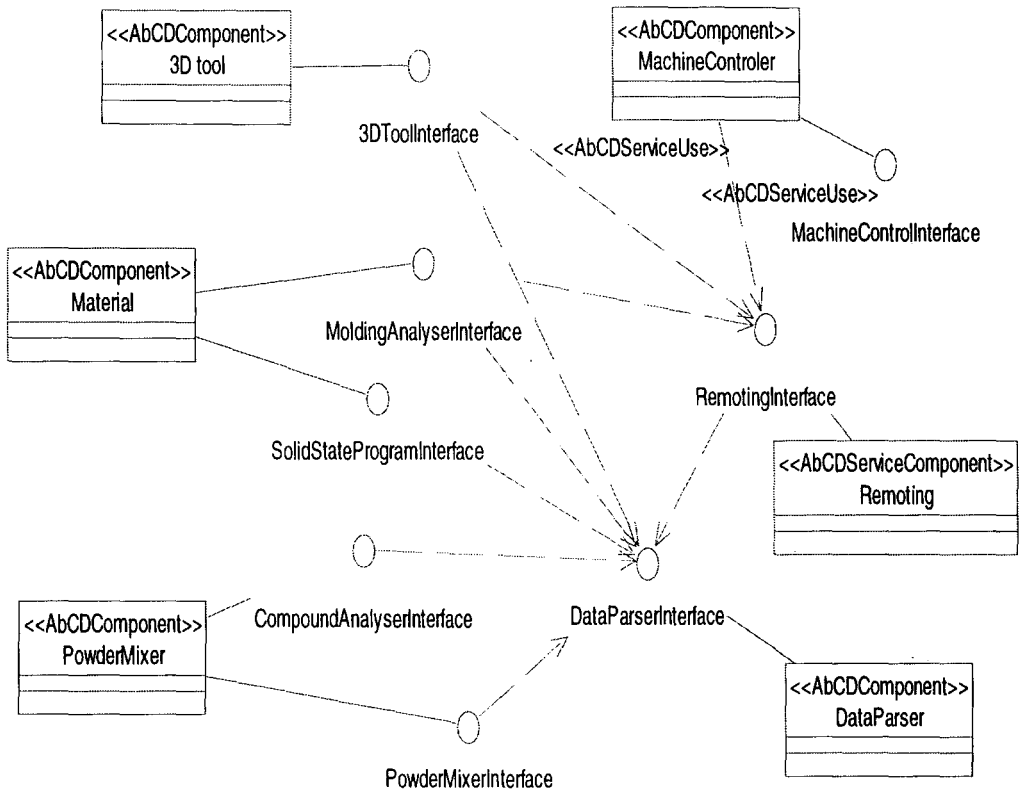


Figure 66 An overview of the AbCD model for RP tool

During this phase, the designers have identified six major logical components to meet functional as well as non-functional requirements. The figure also shows that two extra components are identified. First, the 'remoting' component which can be regarded as a service component, and should be provided by the component framework. Second, the 'DataParser' component which is an explicit identification of the cross-cutting aspect which is required by all other functional components. For each component the designers have added the following requirement aspects as attributes in the component.

- The composition pattern that is requirement by the component to integrate with others.

- The cross-cutting dependency required by the component.
- The service dependency required by the component.
- The packaging and assembly required by the component.

This provides the designers with a logical component with clear functional and non-functional requirements as well as interaction model.

7.3.2.2 Summary of Component construction in RP tool using AbCD approach

Once components are identified, they are constructed using the tool suite. The construction of components allows designers to visualise the dependency model using the component dependency view.

The construction of the logical and abstract AbCD components supports the developers in performing the component acquisition process. This process is achieved by mapping the UML class models generated by the reverse engineering tool to local components. This is different from the construction of UML class for the e-commerce system in case study 1. The AbCD component model has supported the module selection and the decision making process in the following ways:-

- The developers have clear functional requirement of the RP tool.
- The developers have identified the data objects required to integrate amongst different models.
- The developers have a clear cross-cutting dependency view of the components.

One unexpected outcome form this case study is while reverse engineering the source from the modules of different tools, the developers have identified duplicated UML class patterns in the design. This is caused by duplicated code in the source to implement the cross-cutting concerns. Accordingly, the developers were able to re-factor the explicit identification of cross-cutting concerns.

7.3.3 Summary of the Case studies

The two case studies presented in this chapter are used in the evaluation of the AbCD approach, which will be presented in the next Chapter. The comparisons with other related works will also be made on the usability of the modelling approach and reusability of the artefacts that are produced during the design process.

Chapter 8 Evaluation

8.1 Introduction

Chapter 7 described the two case studies that used the Attribute based Component Design approach and discussed the Component Design Guidelines (CDG) introduced in Chapter 4. Having introduced the AbCD meta-model to support the component modelling in Chapter 5, and tool support in Chapter 6, this chapter presents the evaluation work carried out to assess the overall impact on component design and the model driven approach.

The chapter discusses issues relating to how the AbCD approach has an impact on when transforming requirements analysis to design, and from the design to implementation. The evaluation is based on the two case studies described in Chapter 7.

The chapter then presents the evaluation work on the AbCD tool suite which is the implementation of the AbCD approach. The result of the usability of the tool suite is reported. The evaluation on the tools was carried out when applying the case studies.

8.2 The evaluation approach

Before discussing the resulting data, this section describes the evaluation method and the principles applied in the evaluation. The evaluation method carried out in this research is based on the evaluation guideline proposed in literature, namely, DESMET method by [Kitchenham, 1996].

To begin with the evaluation method, the nature of the AbCD approach and the tool developed in the research can be regarded as a non-invasive modelling approach which can assist developers when designing and constructing component based software using any well-defined development process. Accordingly, instead of evaluating with quantitative experiments or surveys, ***a case study based evaluation method*** was chosen because the support provided by the AbCD approach spans three phases of the software development lifecycle. The details of the three phases were already discussed in Section 4.2, when presenting the AbCD approach. The two case studies presented in Chapter 7 provide an opportunity to perform an investigation on the different aspects of the model driven component based development which is shown in Table 11.

It summarise the different aspects that the AbCD approach will have an impact on the process as well as the quality of the modelling artefacts produced when modelling. Quantitative analysis as well as qualitative analysis are made on each aspect. This research also took a position that rely the tool support to apply the method. This is because features proposed in this AbCD approach, such as visualising cross-cutting concerns using component dependency

view can only be achieved by using the AbCD tool suite. Hence, another core effort took place is tool evaluation. It consists of two parts:

- Comparison with other tools
- Feature analysis on the AbCD tool suite

For each evaluation aspect, the chapter presents using the following format.

- the ***aims*** of proposed work (i.e. the object to be evaluated),
- the ***scope*** of the proposed work,
- the ***expected outcome*** of the work,
- and the ***actual output*** of the work.

The structure is used as a framework on the following discussions on evaluation.

Impact	Quantitative effects	Qualitative effects
Impact on the design artefacts	Assessing the artefacts produced using the AbCD approach can improve system (i.e. accuracy, and correctness) and the organisation of the system architecture.	Designers' feedback on usability of the artefacts.
Impact on the design process	Assessing improvements in the developers' workflow for each development phase.	Designer's opinion on whether the approach can accommodate design changes and process improvement.
Impact on the use of the core CbSE design principles when applying the approach	Assessing improvements on core CbSE principles, mainly, abstraction, the use of contract base interface composition, and reusability.	Reviewers' comments on published papers.

Table 11 Evaluation method and inpact areas

8.3 Deriving the AbCD approach: re-addressing the overall 'Aims'

This section re-addresses the overall focus of the research. Therefore the expected outcomes of the aims and the actual outputs can be discussed. The motivation for the initiative of the Attribute based Component Design approach arose out of the problems encountered when designing software based on modern component frameworks. As these frameworks provided a variety of services, as well as standards and interaction models, there is a need to provide an approach that allows developers to construct components in abstract ways. The evaluation work begins with reviewing the ***aims*** set out to be achieved at the beginning of the research.

Initially, the research work started with an aim to propose a meta-model that developers can apply to construct models of software design with an abstraction. The result of this work was the construction of the Attribute based Component Model (AbCD).

Although it is called AbCD, it can be regarded as a modelling approach because it comprises a new AbCD meta-model supported by the Component Design Guidelines (CDG), which were presented in Chapter 4 and Chapter 5 respectively.

To describe the initial rationale behind proposing the meta-model, designing software requires not only the knowledge of the problem domain of the system to be built, but also the implementation details, the technology and the process. Accordingly, the study shows that most software designers and architects are experienced programmers in their chosen programming language and technology. With the emergence of Component based Software Engineering (CbSE), designers with better partitioning of the design into components using component based concepts such as interfaces as contracts, interface based composition, versioning, binary deployment, etc. One of the most exhaustive and detailed study of the CbSE and component concepts were presented by Szyperski in [Szyperski, 1998]. He and other researchers defined a component as a ***unit of deployment*** and a ***unit of third-party composition***. Generally there are generic heavyweight components such as web service or a database, and others based on a particular component framework or technology such as a J2EE component for a .NET assembly component.

This research proposes a different view of a component, as a specification component (i.e. as a design artefact). In other words, a specification component that is independently deployable within the design. Since it is a unit of a design model, the specification component forms an abstraction over implementation components, because it abstracts away from the implementation details of component frameworks.

The following is a summary of the scope of the AbCD approach. It also declares the constraints explicitly.

1. For object-oriented and component based development, UML 2.0 should be used as a core modelling language to apply the AbCD approach. This is because the AbCD meta-model extends the core UML meta-model. However only the subset of the UML, which is the UML class model that represents the static structure of the system, is applied.

2. The approach encourages designers to apply CDG to construct *logical and abstract* components that ensure interactions between components are based on interface composition. CDG is indented to be non-invasive and may be applied using any development process.

3. The AbCD approach provides facilities that allow developers to identify and construct four types of components, data, interface, functional, and service. However the approach is targeted for modelling business components that will be built using a component framework rather than standalone user-interface oriented desktop components. The two kinds of main stream component frameworks aimed to support are heavy weight frameworks such as J2EE, .NET and light weight frameworks such as the Spring application framework.

4. Even though the AbCD meta-model is based on the standard UML meta-model, it is implemented as an Eclipse UML plug-in. Therefore the designer must use the Eclipse IDE tool to be able to apply the AbCD model and the AbCD tool suite to take advantage of the features provided in the tool.

5. The AbCD approach is intended to provide a component dependency view to the OO design using UML that highlights the mapping between functional aspects and component composition.

8.4 Evaluating the artefacts produced from the AbCD approach

The application of the AbCD approach in two different case studies prompted a challenging task. This section discusses the model artefacts produced during the application. The artefacts produced showed that there are tangible (regarded as quantitative) as well as intangible (regarded as qualitative) benefits gains from applying the AbCD approach. In the first case study, the approach was applied when designers were transforming from the analysis model to the actual design of the system. Therefore the main evaluation process took place on assessing how the AbCD approach supported the designers in producing a better quality design.

In the second case study, the approach was applied when the existing design from various projects are analysed for possible re-factoring of the design.

Although there are benefits that have been recognised, many defects of the approach have also been identified when applying in practice. These problems were compounded by the lack of details in the approach and the features of the tool.

8.4.1 Transforming analysis model to specification model

In Chapter 7, the first case study shows how the UML analysis model is transformed into the specification model for the e-commerce application design in case study 1. During the course of applying the AbCD approach to the e-commerce system, a new component specification model was development based on AbCD meta-model. This phase provides an opportunity to investigate the process as well as the resulting artefacts.

The following sections describe the aim, expected outcomes and actual outcomes of new specification model.

8.4.1.1 Non-invasive approach

The aim: **The AbCD approach is intended to be a non-invasive approach**. Accordingly, the new component specification model does not replace the UML design model. Instead it feeds the resulting artefacts back to the system design model. The AbCD approach does not impose a tight process which would allow the designers to apply this approach on any modelling based development process.

Expected outcome: The application of AbCD approach will be integrated to the designers' main modelling process. The artefacts produced in the component specification model will provide a snapshot of the component view in the main design. However designers will require

extra modelling time to follow the Component Development Guidelines proposed in the approach. These include the component identification process and the component construction process. This was depicted in Figure 56. It showed how the construction of the specification component model could be integrated into the two main development processes. For every iteration of transforming analysis to system design, designers are expected construct or improve the component specification model to provide a component view of the design.

Actual outputs : From the evaluation of case study 1, applying the AbCD approach during the analysis to design transformation has significant impact on the design process. The factors influencing on the design process are as follows.

In the initial iterations of transforming the analysis artefacts to design artefacts, constructing a component specification model in every iteration was proven to be unproductive and slows the design process. Feedback from designers depicted that the component specification models constructed were not contributing to the quality of the design.

From analysis, the reason was that during the initial elaboration phases the design was rapidly changing and designers were focusing on capturing only the functional requirements in the design and left out the technical details of the component framework that it is going to be built on. Hence, in the initial iterations they have left out the AbCD approach for constructing the component specification model in the development process. However after a few iterations, and when the functional design became more stable, the AbCD approach was reintegrated with the main development process. For each iteration, designers were identifying components and constructing them using the AbCD approach, which changed *the development direction to component framework driven modelling process*. They were able to apply core CbSE principles to improve the design. These include :-

- better abstraction in design,
- better separation of concerns using component mapping,
- forming design contracts between aspects
- easier re-factoring of the UML class diagrams

Each of the above core values will be discussed in detailed in the following sections.

Another important output from case study 1 was the use UML in the development process. As presented in the literature survey, UML has taken two main roles in design process. Firstly, the use of UML as an informal collaboration medium for designers for understanding of a model or a domain. In this way, models are not documented and normally produced as a

hand-drawn sketch rather than constructed formally using a tool. Secondly, UML as a formal medium where design is driven by UML artefacts produced using a UML tool. Initially, in case study 1, even though designers were applying model driven development, UML was used informally and the models were not documented in every iteration of development. As the AbCD meta-model extends the UML meta-model, the component specification construction process requires the designers to document the UML models in a UML tool to take advantage of the AbCD approach. Even though the aim of the AbCD approach is to be non-invasive in the development process, this constraint has an effect on the modelling process.

To summarise, during the iteration cycles, the AbCD approach should be applied when re-factoring the design that focuses on the infrastructure of the design, partitioning functional, cross-cutting and services provided by component framework that the system will be based on.

8.4.1.2 Abstraction

Hiding complexity by abstraction is one of the most powerful ways to improve design comprehension and modularity.

The aim : The AbCD approach is intended to support the design process by providing abstraction. In other words, the AbCD approach is proposed to resolve the problem of component reusability by allowing designers to construct abstract and logical specification components.

It is achieved by forming a component specification model that captures the functional domain model as well as other cross-cutting aspects of the domain. It is based on the AbCD meta-model that allows the designer to map every object in the design to abstract and logical components. As described in Section 7.3.2.1.1, the logical components have abstracted the objects into functional, cross-cutting, service, and system interface aspects of the design.

Expected outcome : The main expected outcome when applying the AbCD approach is, the designers can exploit the component specification model constructed to partition the design and encapsulate complexity. The AbCD meta-model contains attributes that allow developers to add semantic information regarding technical details for each component. This enables encapsulation over component frameworks' implementation details and standards. However,

the designers must use the AbCD tool suite which is set of plug-ins for Eclipse IDE tool to apply the Meta-model and add the values for various attributes.

Actual outputs: From the result of case study 1, the component specification model provides a power tool for designers.

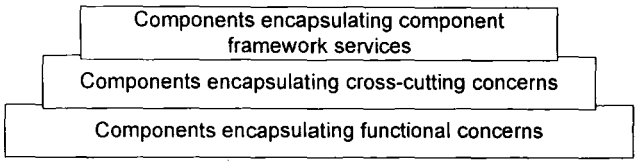


Figure 67 Three layers of encapsulation for the ecommerce system case study

The abstraction provided by the component model is different from traditional abstraction that aims to encapsulate implementation of functionality. As shown in Figure 67, the designers were able construct components that encapsulate three levels of aspects in the design which provide a powerful abstraction from the architecture point of view.

During the initial iterations of the design, although designers were able to encapsulate functional details, they were not able to abstract away all the complexities of object interactions. However for each non-functional service needed in the ecommerce system, designers were able to construct AbCD components that include a ***composition pattern***. These patterns provide an abstraction over how every functional component can interact with service components. For instance, designers have found existing Admin tools (i.e. sending emails, performing backups, etc.), shipping manager and payment manager code to be re-used. Having constructed these components as logical AbCDComponents, they have already defined composition patterns for each component. A composition pattern describes show the component interacts with other components within the system. Hence designers were able to analyse these existing code, to see if these can be adapted to be reused in the system using the composition pattern.

In case study 1, after a few design iterations, designers were able identified the following AbCDComponents that present the functional aspects design of the system.

- Products Manager AbCDComponent
- Customers Manager AbCDComponent
- Supplier Manager AbCDComponent
- Shopping Cart AbCDComponent
- Admin Tools AbCDComponent

- Shipping Manager AbCDComponent
- Payment Manager AbCDComponent

As with most UML models, designers have already partitioned the design into the above aspects. However, transforming the design into AbCD Components captures additional semantic information with abstraction. These include instance management information, remoting interface information and event management information. Furthermore, designers were able to add constraints that should be provided by component frameworks with each service identified. To summarise the output:-

- Designers realised that an abstraction over various services needed by the system provides an opportunity to identify reusable assets.
- Designers were also able to identify potential services that should be provided by the component framework without having to know the detailed implementation.

8.5 Comparison with related works

After evaluating the artefacts produced using the AbCD approach, this section discusses related works published by other researchers. Cheesman [Cheesman and Daniels, 2000] has presented a process for designing components using UML. Cheesman's work focuses on constructing component specification using a proposed process workflow. It proposes a new UML profile and shows how the component models can be derived from business models

In comparison with the AbCD approach, both shares the main aim for achieving abstraction in software component design. However the main differences are as follows:-

- Whilst Cheesman's approach focuses on how to transform business models to component specification, the AbCD approach highlights how non-functional aspects of the business models can be explicitly identified logical components using the AbCD meta-model. This may improves the reusability of the components because the components models are more self-contained.
- Cheesman's approach does not address now components identified can be applied to different technologies for implementation. The AbCD approach highlights how attributes can be used to map platform independent logical component design to platform specific component design for implementation. This improves the usability of the approach as it provides a bridge for PIM and PSM in component design.

Skinner [Skinner, 2001] has presented how UML meta-model can be extended to add the concept of context based attributes to UML modelling. However, Skinner uses meta-modelling by altering the UML meta-model to include new meta-classes. The main limitation of this approach is the tool that implements the UML meta-model also needs to be altered. Skinner showed how the ArgoUML tool is extended to allow designers to add context-based attributes. On the other hand, as the AbCD meta-model is constructed as a UML profile, any UML tool that supports UML profiling can be applied. Using Skinner approach, the designer can add attributes for different generic contexts. However, the attributes provided by the AbCD approach is tailored towards the development of software components that are logical and abstract.

8.5.1 Summary of evaluation on AbCD approach

The AbCD approach will now be evaluated using the core CbSE and MDD concepts identified in the literature survey. Using the table, discussion will be made on how well the AbCD approach fulfils different concepts and solve CbSE problems. The results presented in

Table 12 are based on the two case study work described in Chapter 7. The table also include a grading system as follows :-

- *** Fully supported
- ** Partially supported
- * Minimum support

Concepts	Grade	Discussion
UML Modelling	**	As the AbCD approach only focuses on constructing OO UML model to component model, UML modelling is limited to applying the AbCD meta-model.
UML Profiling for component modelling	***	The profiling tool allows the designer to transform existing UML model to logical component model by applying AbCD meta-model.
Provided Platform Independent Component Modelling	***	The AbCD approach is centered around the concept of presenting components as platform independent.
Provided Platform Specific Component Modelling	**	Although, the approach is targeted for construction of platform independent models, the designer can add platform specific information as attributes.
Specifying non-functional aspects	***	The Graph View tool implemented to support the AbCD approach allows designers to view non-functional aspects included in the design.
Support component selection, filtering and acquisition	***	The attributes in each logical component can be used as meta-data for acquiring existing components for reuse.
Support component Implementation	*	Currently direct no support for component implementation is included in the research.
Support for component assembly	**	The AbCD approach specifies the component design assembly instead of component implementation assembly.

Chapter 9 Conclusion

9.1 Introduction

This chapter reviews the research presented in this thesis and summarises the achievements gathered. It also presents a discussion that highlights the general research contribution to the Software Engineering community. The discussion presented there based on the 7 criteria for success defined in Chapter 1. It also describes the direction for further work.

9.2 Summary of the Research

The problem original problem identified in Chapter 1 was:-

"The component frameworks act as a vehicle for components and takes the responsibility of component management and most importantly it dictates how the component interact using an interaction model. Therefore the designers must have comprehensive knowledge of the particular framework that the design is based on. Accordingly, the architecture of the system design is also dictated by the model supported by the framework. Moreover the component implementation often differs from initial design. This has lead to the position that the component is hard to re-use."

The abstraction in design was identified as the major research problem within this. In a traditional Object Oriented design, abstraction represents encapsulating functional complexity of objects. From this research point of view, abstraction in components hides functional complexity as well as cross-cutting functions, and most importantly the component framework that represents component interaction model and component runtime requirements.

Chapter 2 and Chapter 3 defined the termed component and explored a number of aspects of Component based Software Engineering. In this thesis the termed component has special meaning as defined in Section 2.2.2.

"a software component is a software unit or a building block, which can be independently deployable and composable with other software components, permitting that component contracts are satisfied, and component framework are compatible, to form a component-based system".

It also described three main component frameworks and their associated technologies in detail. This has led to finding the common features and facilities provided by component technologies to form a common abstract and logical framework. This is used as a basis for identifying ways to provide guidelines and a formal approach. This achieved the first criteria for success defined for this research.

> ***"1. Identification of the key factors that improve the quality of design using the core CbSE principles."***

In Chapter 4, the Attribute based Component Design (AbCD) approach was introduced. The approach consists of Component Design Guidelines (CDG) which can be regarded as good practice guidelines. The guidelines are introduced instead of forming a detailed process model because the AbCD approach promotes a non-invasive workflow. The CDG proposed two simple main phases in the workflow: component identification and component construction. This process is enabled by the main focus of this research, the AbCD meta-model.

The detailed specification of the AbCD meta-model was presented in Chapter 5. It allowed designers to model logical components. Each model element represents abstraction over component requirements and composition patterns as attributes of the component. It extends the UML meta-model and proposed 5 new types of logical components covering functional, cross-cutting, data, interface and component assembly aspects of the design. The introduction of the AbCD meta-model accomplished the second criteria for success.

> ***"2. Development of a new meta-model that resolves the problem of component abstraction and allows designers to construct abstract and logical components at specification level."***

The construction of the AbCD meta-model also gave an opportunity to provide visualisation support to view the design in aspects, most importantly to view cross-cutting aspects. Therefore, the AbCD approach provides the Component Dependency View to support the modelling of abstract software components, which satisfies the third criteria for success.

> ***"3. Development of the component dependency view that highlights the cross-cutting and non-functional aspects of the design."***

Chapter 6 presented the implementation work that allows developers to construct AbCD component models using the Eclipse IDE tool. It was constructed as a plug-in to the Eclipse tool, called the AbCD tool suite. It used the existing Eclipse UML 2.0 to derive the AbCD meta-model as a UML 2.0 profile. The tool suite consists of three programs: the profiling tool, analysis tool and the code generation prototype. The implementation work realised the two main criteria for success identified in Chapter 1.

> ***"4. Development of a tool suite that supports the meta-model and enables the component dependency view."***

As part of the evaluation process, the AbCD approach was applied to component based software development projects as two case studies. Chapter 7 presented how the AbCD

approach was applied to the case studies. The two case studies provided two different scenarios introduced in Chapter 7.

> ***"5. Analysis on the productivity of the designers during the development process."***

Throughout Chapter 8 the evaluation work was described based on two case studies. It presented the benefits gained from applying the AbCD approach as well as highlighting the shortcomings of the approach when practising in the case studies. It has been demonstrated that the work presented in Chapter 4 has met the following criteria identified in Chapter 1.

> ***"6. Quantitative and qualitative evaluation of the success of the approach based on two case studies."***

However more work is needed to effectively evaluate the AbCD approach quantitatively, which is the future work.

In Chapter 5, the AbCD meta-model attributes that cover a set of semantics are presented that to support the development of software components. The two case studies presented in Chapter 7 provided an opportunity to assess the richness of the semantics identify and included in the AbCD meta-model for different domains. As presented in Section 8.4, the evaluation shows that the artefacts produced achieved abstraction. However the semantics covered in various attributes of the AbCD meta-model are basic and more improvement is needed to cover a variety of domains. This fulfils the last criteria for success presented in Chapter 1.

> ***"7. Assessment of the rich set of semantics identified by the meta-model to support the design of component based software systems."***

9.3 Future work

The research can be extended in the following ways and some ideas are discussed in this section.

9.3.1 Graphical modelling and tool integration support

Currently, the AbCD tool suite does not support graphical modelling of UML diagrams. Adding graphical modelling will make the designers more productive and also provide visualising support to the model. Furthermore there are no import/export features of the AbCD component model. For instance, adding further functionality to the tool suite in order to allow the designers to import existing UML designs from popular tools such as Rational Rose.

9.3.2 Automating the analysis of the code

Another application of the AbCD approach could be in the field of program comprehension. For instance, a developer might want to understand the design of the existing code. The developer might then try to understand why the different parts of the code are duplicated or clustered. There are methods such as call graph tools, and reverse engineering tools that allow developers to analyse the code. However they do not highlight why code is duplicated. For example, code might be duplicated because of the cross-cutting concerns, forced by the framework that has been built on, or written by inexperienced programmers. The AbCD approach might be able to automatically analysis why parts of the code are duplicated if there is tool that can analyse code. However it might be hard to develop a tool that can analyse duplicated code that understand changes in variables and context but has the same functionality.

9.3.3 Source Generation

Currently, the tool suite includes a simple source generator that generates the necessary configuration file that allows components to correctly configuration to use the Spring application framework. This feature can be extended to include more comprehensive set of source generation facilities to integrate with different component frameworks.

References

(OMG), O. M. G. (1999). "The CORBA Component Model."

Achermann, F. and O. Nierstrasz (2001). Applications = Components + Scripts — A Tour of Piccola. Software Architectures and Component Technology. M. Aksit, Kluwer: 261-292.

Aksit, M. (1996). "Separation and composition of concerns in the object-oriented model." ACM Computing Surveys **28**: 148-148.

Alder, R. M. (1995). "Emerging Standards for Component Software." Computer IEEE **28**(3): 68-77.

Alexander, C., S. Ishikawa, et al. (1977). A Pattern Language.

Alpert, S. R., K. Brown, et al. (1998). The Design Patterns Smalltalk Companion, Addison-Wesley Publishing Company.

Avalon, A. (2005). "Castle Project." from http://www.castleproject.org/index.php.

Beugnard, A., J. Jezequel, et al. (1999). "Making Components Contract Aware." IEEE Computer **32**(7): 6.

Bosch, J. (1996). Composition through Superimposition. Special Issues in Object-Oriented Programming: Workshop Reader of the 10th European Conference on Object-Oriented Programming ECOOP' 96, Heidelberg.

Brown, A. (2004). An introduction to Model Driven Architecture, IBM.

Brown, A. W. (1996). Component-Based Software Engineering. California, IEEE Computer Society Press.

Brown, W. J., R. C. Malveau, et al. (1998). AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, Wiley.

Caldiera, G. and V. R. Basili (1991). "Identifying and Qualifying Reusable Software Components." Computer **24**(2): 61--71.

Canal, C., L. Fuentes, et al. (2003). "Adding Roles to CORBA Objects." IEEE Transactions **29**(3): 242-260.

Chappell, D. (1996). Understanding ActiveX and OLE, Microsoft Press.

Cheesman, J. and J. Daniels (2000). UML Components: A Simple Process for Specifying Component-Based Software, Addison-Wesley.

Cicalese, C. D. T. and S. Rotenstreich (1999). "Behavioral Specification of Distributed Software Component Interfaces." IEEE Computer **32**(7): 7.

Clarke, S. and R. J. Walker (2001). Composition Patterns: An Approach to Designing Reusable Aspects. International Conference on Software Engineering.

Clements, P. C. (1996). "From Subroutines to Subsystems: Component - based Software Development." Component - Based Software Engineering: 2.

Coad, P., J. d. Luca, et al. (1999). Java Modeling Color with Uml: Enterprise Components and Process with Cdrom. Upper Saddle River, NJ, USA, Prentice Hall PTR.

Code-Generation-Network. (2006). "Code Generation Network Website." from http://www.codegeneration.net/generators-by-standard.php?standard=1.

Crawford, W. and J. Kaplan (2003). J2EE Design Patterns Sebastopol, CA 95472, O'Reilly.

D'Souza, D. and A. C. Wills (1999). Objects, Components and Frameworks With UML: The Catalysis Approach. Boston, MA, USA, Addison-Wesley.

Dijkstra, E. W. (1968). "The Structure of the "THE" - Multiprogramming System." Communications of the ACM **26**(1): 49-52.

Dobing, B. and J. Parsons (2006). "How UML is used." Communications of ACM **49**(5): 109-113.

Eclipse (2005). Eclipse UML2

Gamma, E., R. Helm, et al. (1993). "Design Patterns: Abstraction and Reuse of Object-Oriented Design." Lecture Notes in Computer Science **707**: 406-431.

Gamma, E., R. Helm, et al. (1995). Design Patterns: Elements of Reusable Object-Oriented Software.

Garlan, D., R. Allen, et al. (1994). "Architecture Mismatch: Why Reuse Is So Hard." IEEE Software **12**(6): 17-26.

Grundy, J. (2000). "Multi-perspective specification, design and implementation of software components using aspects." International Journal of Software Engineering and Knowledge Engineering **Vol. 10**(No. 6): 713-734.

Harrop, R. and J. Machacek (2005). Pro Spring. Berkely, CA, USA, Apress.

Heineman, G. T. and W. T. Councill (2001). Component-based software engineering: putting the pieces together. Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc.

Hondt, K. D., C. Lucas, et al. (1997). Reuse Contracts as Component Inerface Description. Second International Workshop on Component-Oriented Programming, Turku, Finland, TUCS General Publication.

Hubert, R. (2001). Convergent Architecture: Building Model-Driven J2EE Systems with UML (OMG Press). New York, NY, USA, John Wiley & Sons, Inc.

Jacobson, I. (1993). Object - Oriented Software Engineering. Reading, MA., Addison - Wesley.

Kitchenham, B. A. (1996). "Evaluating software engineering methods and tool part 2: selecting an appropriate evaluation method technical criteria." SIGSOFT Softw. Eng. Notes **21**(2): 11-15.

Kleppe, A., J. Warmer, et al. (2003). MDA Explained: The Model Driven Architecture: Practice and Promise. Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc.

Laddad, R. (2003). AspectJ in Action: Practical Aspect-Oriented Programming. Greenwich, CT, USA, Manning Publications Co.},.

Martin, R., D. Riehle, et al. (1998). Pattern Languages of Program Design, Addison Wesley Longman, Inc.

Mellor, S. J., S. Kendall, et al. (2004). MDA Distilled. Redwood City, CA, Addison Wesley Longman Publishing Co., Inc.

Meyer, B. (1994). Reusable Software: The Base Object-Oriented Component Libraries, Prentice Hall International.

Meyer, B. (2000). Object-Oriented Software Construction, Prentice Hall 2nd edition

Nierstrasz, O. and D. Tsichritzis (1995). Object-Oriented Software Composition. Englewood Cliffs, NJ, Prentice-Hall.

Olafsson, A. and D. Bryan (1996). On the Need for Required Interfaces for Components. Special Issuies in Object-Oriented Programming, ECOOP'96.

OMG. (1998). "Model Driven Architecture." from http://www.omg.org/mda/.

OMG (1999). CORBA /IIOP 2.3 and CORBA Services Specification, OMG.

OMGIDL http://www.omg.org/gettingstarted/omg_idl.htm, OMG.

Orfali, R., D. Harkey, et al. (1996). The Essential Distributed Objects Survival Guide. New York, John Wiley & Sons.

Penix, J. and P. Alexander (1997). Component Reuse and Adaptation at the Specification Level. Workshop on Institutionalizing Software Reuse, Ohio State University, USA.

Pooley, R. and P. Stevens (1999). Using UML Software Engineering With Objects And Components. Edinburgh, Addison-Wesley.

Pritchard, J. (1999). COM and CORBA Side by Side: Architectures, Strategies, and Implementations, Addison-Wesley.

Rashid, A., A. Moreira, et al. (2003). Modularisation and Composition of Aspectual Requirements. Aspect Oriented Software Development Conference.

Rational. (1998). "The Rational Unified Process." Rational Software, The Rational Unified Process, version 5.0, Cupertino, CA, 1998.

Riehle, D. (1997). "Bureaucracy" Pattern languages of program design 3. Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc.

Rogerson, D. (1997). Inside COM. WA, Microsoft Press.

Rumbaugh, J., M. Blaha, et al. (1991). Object Oriented Modelling and Design. Englewood Cliffs, NJ, Prentice - Hall.

Sametinger, J. (1997). Component Interporation. Workshop On Institutionalizing Software Reuse (WISR), Ohio State University, USA.

Schmidt, D., M. Stal, et al. (2001). Pattern Oriented Software Architecture: A System of Patterns, John Wiley and Sons Ltd.

Skinner, M. (2001). Enhancing an Open Source UML Editor by Context-Based Constraints for Components. Berlin, Technical University of Berlin: 10.

Stasko, J., J. Domingue, et al. (1998). Software Visualization: Programming as a Multimedia Experience, MIT Press.

Stein, D., S. Hanenberg, et al. (2002). Designing Aspect-Oriented Crosscutting in UML. AOSD-UML Workshop at AOSD'02, Enschede, The Netherlands.

Suzuki, J. and Y. Yamamoto (1999). Extending UML with Aspects: Aspect Support in the Design Phase. ECOOP Workshops.

Szyperski, C. (1998). Component Software Beyond Object-Oriented Programming, Addison Wesley Longman Limited.

Vigder, M. R. and J. Dean (1996). COTS Software Integration: State of the art. Ottawa, Ontario, Canada, Institute for Information Technology.

Warmer, J. and A. Kleppe (2003). Object Constraint Language 2nd Edition, Addison Wesley Professional.

Zaremski, A. M. and J. M. Wing (1997). "Specification matching of software components." ACM Transactions on Software Engineering and Methodology **6**(4): 333-369.