Meta Level Component-Based Framework for Distributed Computing Applications

ANDY SHUI-YU LAI

Doctor of Philosophy

ASTON UNIVERSITY

April 2008

THESIS SUMMARY

ASTON UNIVERSITY

Meta Level Component-Based Framework for Distributed Computing Applications

ANDY SHUI-YU LAI

Doctor of Philosophy
April 2008

Adaptability for distributed object-oriented enterprise frameworks is a critical mission for system evolution. Today, building adaptive services is a complex task due to lack of adequate framework support in the distributed computing environment. In this thesis, we propose a Meta Level Component-Based Framework (MELC) which uses distributed computing design patterns as components to develop an adaptable pattern-oriented framework for distributed computing applications. We describe our novel approach of combining a meta architecture with a pattern-oriented framework, resulting in an adaptable framework which provides a mechanism to facilitate system evolution.

The critical nature of distributed technologies requires frameworks to be adaptable. Our framework employs a meta architecture. It supports dynamic adaptation of feasible design decisions in the framework design space by specifying and coordinating meta-objects that represent various aspects within the distributed environment. The meta architecture in MELC framework can provide the adaptability for system evolution. This approach resolves the problem of dynamic adaptation in the framework, which is encountered in most distributed applications. The concept of using a meta architecture to produce an adaptable pattern-oriented framework for distributed computing applications is new and has not previously been explored in research.

As the framework is adaptable, the proposed architecture of the pattern-oriented framework has the abilities to dynamically adapt new design patterns to address technical system issues in the domain of distributed computing and they can be

woven together to shape the framework in future. We show how MELC can be used effectively to enable dynamic component integration and to separate system functionality from business functionality. We demonstrate how MELC provides an adaptable and dynamic run time environment using our system configuration and management utility. We also highlight how MELC will impose significant adaptability in system evolution through a prototype E-Bookshop application to assemble its business functions with distributed computing components at the meta level in MELC architecture. Our performance tests show that MELC does not entail prohibitive performance tradeoffs. The work to develop the MELC framework for distributed computing applications has emerged as a promising way to meet current and future challenges in the distributed environment.

Keywords:
Component-based software, frameworks, distributed computing, design patterns, meta architecture.

Acknowledgements

This is to declare that the work of this PhD thesis was done by me and the work has not been submitted for any other academic award.

My thesis supervisor, Dr A J Beaumont, and his colleagues, Dr E F Elsworth and Mr B S Doherty, have provided invaluable advice and comments on the work shown in thesis. I would especially like to thank my supervisor for providing encouragement throughout the past six years. I would like to give a special mention to my examiners, Dr D Cornford and Dr B Bordbar, for their comments and supports to my work.

In particular, I wish to extend special thanks to my External Supervisor, Dr Y K Leung, Head of the Department of Information and Communications Technology, Hong Kong Institute of Vocational Education, to provide me with valuable ideas and allowing me sufficient time to write up the thesis. Mr. Jacob Chu also deserves special mention for his help in editing and compiling the manuscript.

I also thank my students, Marco Chan, Harry Lee, and many anonymous students, who help to prepare the program coding and conduct testing in the feasibility study on the proposed adaptable MELC framework.

And finally, I would like to thank my wife Esther and my son David for their love and continued support throughout the years.

Last but not least, I would like to thank the Hong Kong Vocational Training Council for providing me Staff Development Fund (#SDS000163) to complete my PhD study at Aston University.

Table of Contents

PART III ADAPTABLE ARCHITECTURE OF MELC

List of Figures and Tables

PART I

INTRODUCTION

Chapter 1 Introduction

1.1 Introduction

Many application designers aim to create reusable components in their applications to reduce the effort and time required for software development. Reusable components range from simple classes and libraries to reusable patterns and frameworks.

The level of component reuse in the development life cycle depends on the nature and granularity of the component and the decisions made by the component designer on behalf of the component user. A simple design decision, like the reuse of a library class or an API, is the most common and represents reusability at the class level.

Design patterns operate at a higher level of abstraction than classes and provide a successful way of describing communication between classes. In fact, application designers probably don't write any code until they can build a picture in their mind of what code does and how the pieces of the code interact. The more they can picture this organic whole, the more likely they are to feel comfortable that they have developed the best solution to the problem. In one sense, the more elegant solution will be more reusable and more easily maintainable. Design patterns satisfy this need for good, simple, and reusable solutions.

A design pattern is a convenient way of reusing object-oriented code between projects and between application developers. The idea behind design patterns is simple: to catalog common interactions between objects that application developers have often found useful.

Patterns support a problem-solving discipline with acceptance in the software architecture and design community. Patterns represent recurring solutions to software development problems within a particular context. A pattern is a structured document that describes a solution. It captures the key design constructs, practices, and mechanisms of core competence in object-oriented development.

Recently, design patterns were introduced to document good OO designs [1, 2]. In general, a pattern describes a problem that frequently occurs in software application, and then describes its solution in such a way that it can be reused. Design patterns help designers reuse successful designs by basing new designs on prior experience.

In building design frameworks, we focus on reusing design patterns. It is clear that less emphasis has been placed on systematically deploying these reusable designs than on documenting them. Reusing software in practical applications is a difficult task, yet it is required to reduce development effort and assure high software quality. Reusable designs and frameworks are less frequently used due to the complexity and difficulty of constructing generic designs for common application domains.

There are framework developers in more complex domains (such as multimedia networking framework [3], adaptive web server framework [4], E-commerce framework [5] and Grid-based Flood Monitoring Framework [105]). As a result, developers in these domains largely build, validate, and maintain software systems from scratch. In an era of deregulation and stiff global competition, however, it has become prohibitively costly and time consuming to develop applications entirely in-house from the ground up. The construction of a generic framework using design patterns for developing distributed computing

applications has hitherto received insufficient emphasis in research.

The approach of using design patterns in constructing a framework makes its design easier to understand. In addition, the framework thus generated contains better software quality [6]. However, the discussion of pattern-oriented framework construction shows that building frameworks based only on patterns has not received much research attention, because of difficulty in maintaining the integrated coding after the instantiation of design patterns, though benefits of patterns such as reusability and modularity are well known and understood [6]. However, it has been observed that pattern-oriented frameworks lack the adaptability to enable system evolution [85].

But our work in this thesis demonstrates the use of reflection with a meta architecture to resolve the adaptability problem in a pattern-oriented approach. On one hand, we aim to apply the concepts of a meta architecture to a pattern-oriented framework to resolve the difficulty in maintaining the integrated coding after the instantiation of design patterns and, on the other hand, treat instances of patterns as components that will accommodate and adapt to the ever-changing system evolution. We have observed that the components in distributed applications require runtime replacement to meet the system evolution in a distributed setting.

In this thesis, we introduce a new architecture for the pattern-oriented framework that has the ability to dynamically adapt new design patterns to address issues particularly in the domain of distributed computing. In effect, we augment the adaptability of current reflective frameworks in order to support runtime component replacement that is required in system evolution.

In particular, we view the system functionalities of a distributed architecture as encapsulated collections of components. We employ distributed computing

patterns as building blocks for meta components as part of our meta-based architecture.

The remainder of this thesis is organized as follows.

Part I - Introduction

Our meta-based framework will rely heavily on object-oriented (OO) design features like abstraction and dynamic binding to achieve extensibility and adaptability. Chapter 2 discusses the relationship between Design Patterns and Framework and presents the key properties of an adaptable framework for distributed computing applications.

Part II – Technical Aspects and Related Works

Chapter 3 discusses recent works on the use of components. Chapter 4 introduces a *pattern-oriented* framework, which uses design patterns as building blocks. The *pattern-oriented* framework satisfies the need for a good and reusable solution but is not able to provide adaptability for system evolution. In Chapter 5 and Chapter 6, we show the related works (theoretical works and practical works) of adaptability in meta architecture.

Part III – Adaptable Architecture of MELC

Chapter 7 describes the architecture of our Meta Level Component-Based Framework (MELC). We describe our design and implementation which combines a meta architecture with a pattern-oriented framework. The result is an adaptable framework which provides a mechanism to facilitate system evolution. An adaptable layer is put into a pattern-oriented framework by using a meta architecture creating a higher level of abstraction. The MELC Programming Model is also introduced. Chapter 8 summarizes the common issues in distributed computing and provides resolution for them with their relevant design patterns. Chapter 9 shows how to build an application with MELC.

Chapter 10 describes how MELC instantiates design patterns into meta components. Chapter 11 describes, in detail, the implementation of adaptability in MELC. It covers the replacement of meta objects at runtime and their implementation with the programming model.

To demonstrate that the MELC framework can be implemented efficiently, Chapter 12 conducts an analysis of the framework performance. Finally, Chapter 13 provides a conclusion, highlighting MELC' s principal contributions to distributed computing.

Chapter 2 Requirements for Adaptable Framework

2.1 Frameworks

Our aim is to develop a component-based framework which uses distributed computing patterns as components in that framework. The advantages of using design patterns and a framework is that design patterns and frameworks both facilitate reuse by capturing successful software development strategies [15]. The primary difference is that frameworks focus on reuse of concrete designs, algorithms, and implementations in a particular programming language. In contrast, design patterns focus on reuse of abstract design and software micro-architectures [20].

Frameworks are an object-oriented reuse technique [11]. A framework is a reusable design of a system that describes how the system is decomposed into a set of interacting objects. Sometimes the system is an entire application; sometimes it is just a subsystem. The framework describes both the objects and how these objects interact. It describes the interface of each object and the flow of control between them. It describes how the system's responsibilities are mapped onto its objects [21]. Frameworks take advantage of all three of the distinguishing characteristics of object-oriented programming languages: data abstraction, polymorphism, and inheritance [7][8].

2.1.1 Our Definition of a Framework

Yacoub [6] described framework using a pattern-oriented approach. He views a framework as a concrete realization of families of design patterns that are targeted for a particular application-domain. When patterns are used to structure and document frameworks, nearly every class in the framework plays a

well-defined role and collaborates effectively with other classes in the framework. We adopt his definition and apply it in this thesis.

2.1.2 *Other Definitions of Frameworks*

Many authors have different interpretations of what constitutes a framework. These definitions are not conflicting, rather, they describe several aspects of frameworks such as structure (how it is constructed), instantiation (how it is used), and classification of frameworks (black box / white box or application / domain-specific). Generally, a framework is a frame on which something will be built.

Fayad et al. [20] present a comprehensive discussion on OO frameworks in which they classify application frameworks and discuss their general strengths and weaknesses.

Johnson et al. [22] describe a framework as a reusable "semi-complete" application that can be specialized to produce custom applications. They discuss classifications of white-box and black-box frameworks. In black-box frameworks, the source code of the original framework cannot be modified, only extended [23], while white-box frameworks require understanding of the frameworks structure and the hot spots to which application-specific functions are hooked.

D'Sonza [24] refers to framework as "a pattern of model or code that can be applied to different problems" and further refers to OO frameworks as "collaborations with a default, skeletal implementation."

Schmid [25] classifies frameworks as being either application- or domain-specific. Application-specific frameworks provide the basic functionalities of a working

application, but the specific contents that model the application domain have to be added for each particular application. Domain-specific frameworks are less commonly used; they model the domain-specific functionality using common objects and generic application logic that can be found in a particular domain. Configuring these objects and binding the generic application logic to the configuration builds an application.

Rogers [26] describes a framework as "A class library that captures patterns of interaction between objects, ... Consists of a suite of concrete and abstract classes, explicitly designed to be used together. Applications are developed from a framework by completion and implementation of abstract classes."

Bushmann et al. [2] define frameworks in an architectural context as: "A partially complete software system that is intended to be instantiated. It defines the architecture for a family of systems and provides the basic building blocks to create them. It also specifies the places where adoptions for specific functionality should be made. In an object oriented environment a framework consists of abstract and concrete classes."

2.1.3 *Benefits of Using Frameworks*

It has been observed that the primary benefits of object-oriented frameworks stem from the modularity, reusability, extensibility, and inversion of control they provide to software developers [12]. The architectural level, instantiated level and applicable level of different stages in framework development cycle provides many different software qualities to a framework, as illustrated in Figure 2.1 and described in the following:

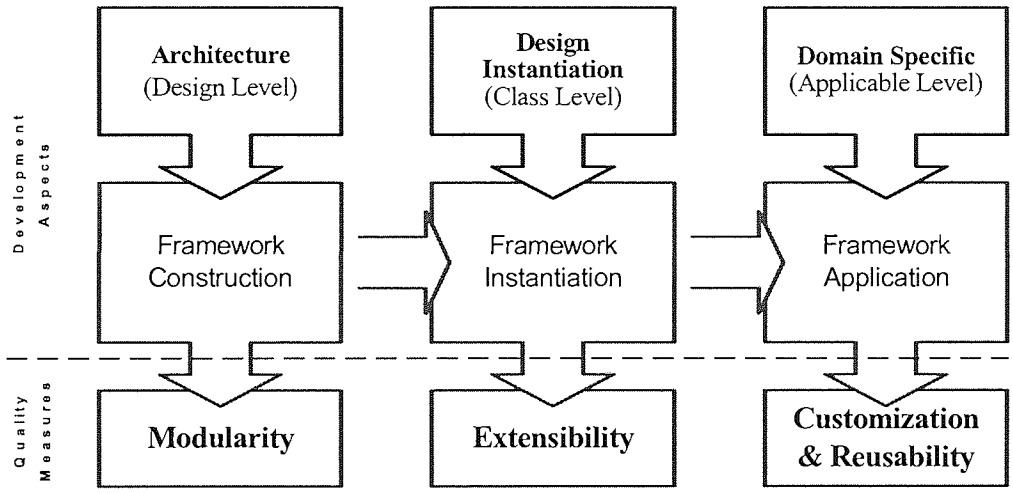

Figure 2.1 Framework Aspects and Quality Measures

Modularity – Framework construction with object-oriented approaches enhances modularity by encapsulating volatile implementation details behind stable interfaces. Framework modularity helps improve software quality by localizing the impact of design and implementation changes [16]. The localization reduces the effort required to understand and maintain existing software.

Reusability - The stable interfaces provided by frameworks enhance reusability by defining generic components that can be reapplied to create new applications [20]. Framework reusability leverages the domain knowledge and prior effort of experienced developers in order to avoid recreating and revalidating common

solutions to recurring application requirements and software design challenges. Reuse of framework components can offer ease of use to programmers, as well as enhancing quality.

Extensibility – A framework enhances extensibility by providing explicit hook methods (further discussed in the next section) that allow applications to extend its stable interfaces [27]. Hook methods systematically decouple the stable interfaces and behaviors of an application domain from the variation required by instantiations of an application in a particular context. Framework extensibility is essential to ensure timely customization of new application services and features.

Inversion of Control – The runtime architecture of a framework is characterized by an inversion of control. Inversion of control allows the framework (rather than each application) to determine which set of application-specific methods to invoke in response to external events.

2.2 Applying Design Patterns for Framework Development

Design patterns can be applied to framework design as well as other software development. In this section, we illustrate the benefits of using design patterns to build a framework and, in fact, we apply the same technique as the fundamental groundwork to build our framework. The concept of abstraction in object-oriented software development is described in the books at various levels of detail, such as Eriksson [10]; Jacobson, Booch, and Rumbaugh [11]; and Pooley [12]. The use of abstraction in object-oriented software development separates interfaces from classes. It is common to have both an interface and an abstract class defined in the same package. In addition to providing an interface, an abstract class provides part of the implementation of its subclasses.

Patterns apply to framework design just as to any other software solution. For example, a *template method* design pattern defines the skeleton of an algorithm in an abstract class, deferring some of the steps to subclasses [16]. Each step is defined as a separate method that can be redefined by a subclass, so a subclass can redefine individual steps of the algorithm without changing its structure. The abstract class can either leave the individual steps unimplemented or provide a default implementation. The class diagram of the structure of *template method* design pattern is depicted in the Figure 2.2.

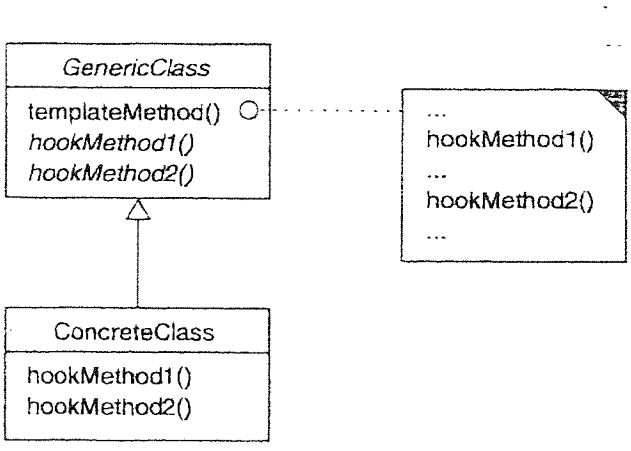

Figure 2.2 Class Diagram for Template Method Design Pattern

The participants in the *Template Method* design pattern are the

1 `GenericClass`, which defines abstract *hook methods* that concrete subclasses override to implement steps of an algorithm and implements a template method that defines the skeleton of an algorithm which calls the hook methods;

2 `ConcreteClass`, which implements the hook methods to carry out subclass specific steps of the algorithm defined in the template method.

In the *Template Method* design pattern, *hook methods* do not have to be abstract. The generic class may provide default implementations for the hook methods. Thus the subclasses have the option of overriding the hook methods or using the default implementation.

2.2.1 *Use of Template Method Design Pattern for Framework Development*

The following section tries to illustrate the use of abstract classes and the *Template Method* design pattern to implement domain specific frameworks and demonstrates that *Template Method* is reasonable solution to construct a framework.

In this example, we design and implement a generic function plotter applet, *Plotter*, for plotting arbitrary single-variable functions on a two-dimensional canvas [13]. The generic plotter should factorize all the behavior related to drawing and leave only the definition of the function to be plotted to its subclasses.

A concrete plotter `PlotSine` will be implemented to plot the function

$$y = sin\ x$$

The *template method* pattern offers a reasonable solution. The single-variable function to be plotted can be represented as a *hook method* in the generic plotter class.

The function plotting method can then be defined in the generic plotter classes as a template method. The generic plotter class is outlined in the following Java program.

```
public abstract class Plotter {

      //the hook method
      public abstract double func (double x) ;

      //the template method
      protected void plotFunction (Graphics g) {
            ....
            func( double x);
            ....
      }
}
```

The concrete plotter will only need to extend the generic plotter and define the function to be plotted by overriding the hook method. The program structure is shown below:

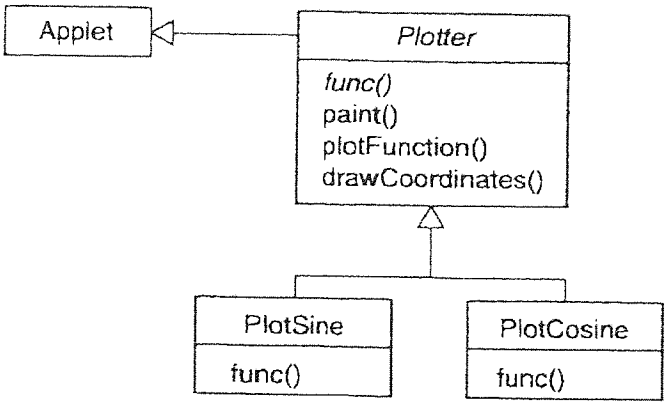

Figure 2.2 *Plotter* as Generic Class and *PlotSine* and *PlotCosine* as Concrete Classes

2.2.2 *Implementation of a Generic Framework*

The method `plotFunction` is the template method in the Template Design Pattern.

```
protected void plotFunction(Graphics g) {//the template method
  for (int px = 0; px < d.width; px++) {
    try {
      double x = (double)(px - xorigin) / (double)xratio;
      double y = func(x);
      int py = yorigin - (int) (y * yratio);
      g.fillOval(px - 1, py - 1, 3, 3);
    } catch (Exception e) {}
  }
}
```

`PlotSine` and `PlotCosine` are concrete classes of `Plotter` class. They simply extend the `Plotter` class and implement the hook method `func()`.

```
public class PlotSine extends Plotter {
  public double func(double x) {
    return Math.sin(x);
  }
}
```

and

```
public class PlotCosine extends Plotter {
  public double func(double x) {
    return Math.cos(x);
  }
}
```

This pattern solves a specific design problem for graphical plotting and makes object-oriented designs in framework architecture more flexible, elegant, and ultimately reusable. A designer who is familiar with this pattern can apply it immediately to design problems without having to rediscover them.

Framework applications described in the previous section consist of framework layer and application layer. The generic `Plotter` *class* is part of a framework

layer that can be used for creating different versions of a graphical display for a java applet, whereas the `PlotSine` and `PlotCosine` classes are in the application layer and are used in specific applications.

In the program `Plotter.java`, the `plotFunction` method is a template method, a frozen spot in the framework layer, and the `func` method is the hook method (a hot spot) for a concrete subclass in the application layer [16]. The frozen spots (template methods) stay hidden in the classes of a framework layer and are implemented with fixed operation, whereas the hot spots (hook methods) are implemented in concrete subclasses in the application layer, an example being the implementation of the `func` method in the subclasses `PlotSine` or `PlotCosine` of class `Plotter`.

In the Plotter example given above, the operational flow for class `PlotSine` is controlled by class `Plotter` in the framework layer. The operational flow is controlled by the framework layer and the application layer provides the functions called by the framework. The framework will inversely control applications in the application layer.

It has been observed that this plotting example is particularly important. The instances of `PlotSine` and `PlotCosine` in the framework are interchangeable at the object level. Both `PlotSine` and `PlotCosine` are subclass of `Plotter`. Every instance of `PlotSine` or `PlotCosine` is an instance of `Plotter`, and everything that applies to `Plotter` also applies to instances of `PlotSine` or `PlotCosine`. The hook method, `func`, causes the functionality in `PlotSine` and `PlotCosine` to behave differently when they are used for plotting. We adopt this design pattern, *Template Method*, as fundamental groundwork for developing our framework.

Consider a system function in distributed computing environment, *Thread Pooling*, shown in Figure 2.3. The inheritance hierarchy shows different levels of abstraction of the abstract class, *ThreadPooling*, whereas its subclasses represent various specialized abstractions, in our cases, they are *Fixed Thread Pool* and *Growthable Thread Pool*. They can be interchangeable as they are implementation of the same abstraction, *ThreadPooling*, in the framework.

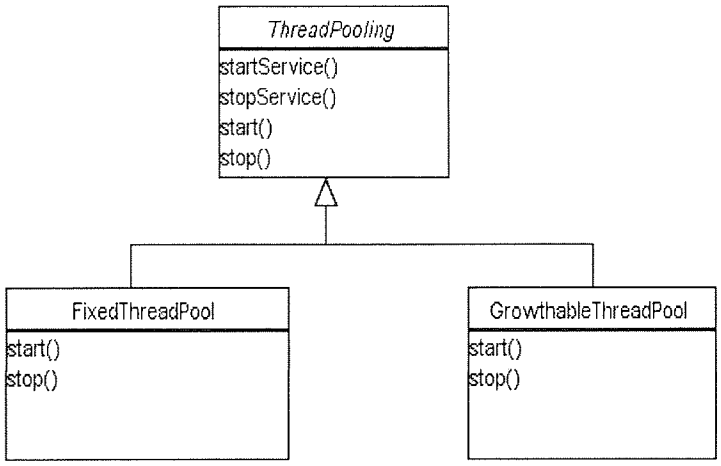

Figure 2.3 *ThreadPooling* as Generic Class and *FixedThreadPool* and *GrowthableThreadPool* as Concrete Classes

The *Template Method* design pattern is the most commonly used pattern to develop framework [16]. We adopt this design pattern, *Template Method*, as a basis for developing our framework. However, applying *Template Method* design pattern for framework development has limited the framework to the *template methods* in a class in the framework layer. It is only good for developing interchangeable objects of an inheritance hierarchy in simple framework architecture. But for a reflective and adaptable component-based framework like the one we proposed, it will be extremely difficult to manage the objects integration and coordination in a complicated component-based framework architecture, and has difficulties in providing runtime adaptability for object

replacement in the distributed computing environment for system evolution. In this thesis, we adopt *Template Method* as the fundamental groundwork to build our framework and apply other software techniques on top to overcome the difficulties identified.

2.3 Properties of Adaptable Distributed Computing Framework

The accelerating pace of change in distributed computing technology confronts software developers and compels them to make their systems more configurable, flexible, and adaptive [85]. The unavailability of adaptation of critical systems in the real world, such as web services, telecommunication switches and banking systems could have unacceptable consequences for the institution's environment. These systems can not be easily taken offline for maintenance due to high costs of their down time (telephone and banking), environment safety (nuclear plants) or loss of human life (life support systems). These systems are difficult to upgrade [103]. An increasingly important requirement of software systems is adaptability to continuous evolution. For the mission-critical systems, adaptability is essential.

Software systems with contemporary distributed computing technology evolve to face unexpected situations or just for self-improvement. On the other hand, software engineering disciplines of current object-orientation can not cope with all situations, particularly with regard to evolving, non-stopping systems.

It has been observed that distributed applications essentially require a software framework architecture that will enforce five important properties [85]: *separation of concerns, adaptability, transparency, extensibility* and *portability*. Such an architecture we call as "adaptable" distributed computing framework because its implementation is required to support adaptation for system evolution. Further, it provides flexibility such as incremental development in the

implementation of software components.

Each property of such adaptable framework architecture for distributed computing applications is listed below:

Separation of Concerns

Separation of Concern is a software design tool for software modularization. The underlying philosophy is to break down a problem into smaller parts. If the framework architecture adopts this principle for architectural decomposition, then each module solves or implements some distinct set of concerns in their applications. Such a module is used in our framework as a component. Separation of Concern enables the framework architecture to start design from high-level abstraction in the development process and partition the functions into framework components. The use of separation of concern technology provides benefits of innovative composition and consequent maintenance to framework architecture.

Adaptability

Adaptability in the framework architecture refers to the ability to which the architecture can dynamically adapt to new user requirements or newly invented technologies. This is achieved through inserting or replacing software modules or system functions at runtime while the rest of the system is still running and the behavior of the framework reflects the changes. Such reflective framework architecture should be flexible enough to meet a wide range of requirements on demand; not only will the framework allow the insertion of modules; it should provide opportunities for the replacement of the established modules after instantiation. The modules in the framework have no cohesive dependency, so that they are not firmly bound together while allowing themselves to evolve independently

Transparency

Frameworks aim at providing services for distributed enterprise applications. It embraces lightweight technology (such as ORB for Object Request Broker) which transparently hides distribution and other non-functional concerns from software developers. Software developers usually prefer not to write the coding for communication between computers but rather, between objects, as happens in object-oriented programming. In the distributed object environment, the communication from clients and servers is designed through distributed objects. An object broker is an intermediary in interactions between clients and servers.

Currently, distributed applications are run on several computers at different locations with all communication between computers in different environments restricted to messages. The gap between the representation of data at the programming level and at the physical level is closed by protocols, a kind of linguistic convention between the individual devices involved in data transmission. The framework architecture should be transparent to software developers in distributing objects between computers, with both middleware and applications being built under the lightweight component model [29].

Extensibility

One of the major requirements of framework architecture is to be open-ended for extension, whereby it opens up the infrastructure (internal machinery) with all system components residing inside. For example, the object broker in lightweight component technology is one of the system components, which in turn is the middleware for the object distribution in the framework. Software developers will be able to engineer the component and its configuration in the framework without altering a language's compiler or extending interpreter. The architecture will expose its components to software developers and allow the software

components to be plugged and integrated without restarting application servers. In addition, the framework architecture should possess the ability not simply to support the anticipated adaptive requirements but also unanticipated requirements of system evolution in the future as well.

Portability

Framework architectures should not be bound to a particular language. The design of framework architecture is portable and is independent of programming language and platform. Furthermore distributed services in the framework should be provided via data communication, with independent interfaces and network transport protocols. Thus, the portability is one of requirements in the adaptable framework architecture to support the independence in the implementation of the architecture and the data transmission in network transport.

Portable framework architectures should not be built in such a way as to add adaptability to a compiled language such as C++ and Java, by extending the language interpreter. If indeed the language interpreter was extended, these framework architectures would have strong cohesion with implementing languages, and their own flexibility decreased, such as the version changes of the compiled language (e.g. JSDK 1.3 to JSDK 1.4) required the extension of the language interpreter accordingly. The adaptability of distributed services in the framework architecture should be transferable between versions in software evolution [82].

2.4 Summary

Framework architectures strive to accelerate the building of distributed components by providing extensive sets of services and a runtime adaptable framework to manage the services for software evolution.

In this chapter, we have identified mission-critical applications that are in need of adaptability for continuous system evolution and also identified five important properties of an adaptable distributed computing framework. They are: *separation of concerns, adaptability, transparency, extensibility,* and *portability.*

This thesis aims to provide an adaptable distributed computing component-based framework. In Chapters 3 through 6, the properties in the adaptable distributed computing framework and their aspects will be used to facilitate further discussion of their related works and applications to the architecture of our adaptable framework.

Chapters 7 to 13 will thoroughly illustrate the architecture design and evaluation of our proposed framework. They will demonstrate how our adaptable framework proposed in this thesis can meet the challenges of supporting adaptability in distributed computing framework while covering the five important properties identified in this chapter.

We show that the organization and structure of the thesis logically presents conceptual modeling and the physical implementation of our framework. It demonstrates how our framework can make significant contributions towards adaptable architecture for distributed computing systems.

PART II

TECHNICAL ASPECTS AND RELATED WORKS OF MELC

Chapter 3 Components and Component-Based Development

Our work in this thesis aims to develop components which are well separated from their environment and other components, and are functionally self-contained. Such that components can be developed each in isolation. We apply the *Separation of Concerns* (SOC) technology for component development for architectural decomposition as well as concern modularization, and *Separation of Concerns* is one of the key properties of our adaptable framework. The domain specific classes and groups of these classes with a defined interface can constitute a domain specific component. The concerns of system functions and groups of such concerns can also constitute a component. The framework we proposed in this thesis adopts a new development process which differs from the traditional waterfall model. The task of building a system by writing code in the traditional model has been replaced with building an application system by developing good quality and reusable software components, and assembling and integrating existing software components.

3.1 Components

Components are defined in various ways from similar and different points of view. But to come up with a precise and well-understood definition of a component, which everybody agrees upon, is not an easy task. In this section, a compilation of definitions regarding components is given, which clarifies the differences in the relation between objects and components.

3.1.1 Our Definition of a Component

Alexander [43] gives the most desired properties of components and we adopt his

definition and apply it in this thesis. Alexander defines component as "A component should be able to plug and play with other components so that it can be composed at run-time without re-compilation. Moreover, components should separate their interface from their implementation. Furthermore, components should be able to inter-operate on a pre-defined architecture. Finally, component interfaces should be standardized so that they can be widely reused."

According to Alexander's definition, software developers are able to cleanly separate the different concerns in components of some kind and develop each in isolation. They are separation of concerns in conceptual modeling. However, some challenges are inevitable in a component-based software development. They are components interaction/communication with client, component co-operation in a framework architecture, and the co-ordination embodied with other components in a component framework.

3.1.2 Other Definitions of Components

Some other definitions of the component have emerged in the literature:

In a COM technical review from Microsoft [41], a component is a piece of compiled software, which offers services. This property is obviously true for components, but is too weak for a definition. It is even applicable to compiled libraries. This means that a component is the minimal piece of functionality. According to this definition, components have to be minimal. A composition of components is not acceptable.

Gamma [1] describes a component as a unit of independent deployment, which is a unit of third-party composition, with no persistent state. This means that a component is well separated from its environment and other components, and is

sufficiently self-contained. It should come with clear specifications of what rationale and intent is and what problem it addresses. The specific features like component integration and component co-operation are not covered.

Eden [42] defines a component as a collection of co-operating objects, with a clearly defined boundary separating them from other objects or components. Objects inside a component typically are intertwined tightly, while the interaction across the component boundary is relatively weak.

3.2 Objects and Components

Objects provide the basic elements for a component. Typically, a component comes to life through objects and thus would normally contain one or more classes. Objects and classes have a number of properties in common with components. Most of the ideas developed in traditional and object-oriented programming theory will be usable in component-based software development as well. They are inheritance, aggregation and interface. In addition, objects are reusable entities that could be connected together into programs. We know, however, a number of properties more relevant to components:

1 A component does not have to be an object. It can be a domain specific function, or even an executable program but that cannot be treated as an object [44].

2 A component's access to implementation from outside of the component is prevented. Objects within a single component may access each other's implementation, but not accessing implementation of other components [45].

3 A component may be distributed and used from within different

programming environments [46]. Object-orientation binds the implementation to a particular class library and language. Components are not bound to a particular language and they communicate through independent interfaces and network transport protocols [44].

4 A component may be constructed by a third-party vendor known as commercial off the shelf (COTS) and its source code may not be available for evaluation. [46].

A component is not an object. The life cycle of the component software development process is different to object-orientation, whereas the component depends on the component framework architecture. Components are often large-grained and have complex actions at their interfaces. It is particularly important that the component-based framework establishes environmental conditions for the component instance and regulates the interaction between component instances.

3.3 Components in Distributed Applications

Distributed computing applications execute on geographically distributed nodes supported by a local or wide area network. Client/server applications, distributed real-time data collection applications, and distributed real-time control applications are all examples of distributed applications.

A distributed application is structured into distributed subsystems. Each subsystem is designed as a configurable component and corresponds to a logical node. Thus, a subsystem component is defined as a collection of concurrent tasks executing on one logical node [106]. However, because more than one subsystem component (logical node) may execute on the same physical node (workstation),

the same application could be configured to have each subsystem component allocated to its own separate physical node (workstation) or to have all or some of its subsystem components allocated to the same physical node (workstation).

The design of message communication interfaces includes asynchronous communication, synchronous communication, client/server communication and brokered communication. They are the common message-based design for subsystem components in distributed computing and their technical details are explained in Appendix D.

An important goal in the design of software architecture for a distributed application is to provide a concurrent message-based design that is highly configurable on one logical node. In other words, the objective is that the same software architecture should be capable of being mapped to many different system configurations. A component-based development approach, in which each subsystem is designed as a distributed self-contained component type, helps achieve the goal of developing a distributed framework to have a distributed, highly configurable, message-based design.

A distributed component has a well-defined interface. A *component* is usually a composite object composed of other objects. A *component type* is self-contained and thus can be compiled separately, stored in a library or a package, and then subsequently instantiated and linked into an application [29]. A well-defined *component type* is capable of being reused and instantiated in different applications from that for which it was originally developed [107]. As a matter of fact, the system-specific distributed computing design patterns (e.g., Thread Pooling, Heart Beat, ··· etc) are candidates to be selected to be the component types in a distributed computing environment. They can be realized in classes stored in packages that are the self-contained domain specific component types that can be reused and instantiated in different distributed applications. The

component types implemented in MELC are system-specific distributed computing design patterns [111] and they are the building blocks for the construction of the framework.

3.4 Concerns, Design Patterns and Components

3.4.1 Separation of Concerns (SOC)

A software design concern can be basically any cognitive element that can be considered while building a program (e.g., a protocol, a feature, a requirement etc.) [47]. In the process of decomposition of an application into program units, typically some concerns remain scattered across the decomposed program units. For example, if a client and a server unit communicate using some form of protocol, it might prove very difficult to cleanly factor out the code that implements the protocol as a separate program unit.

The scattering of concerns also implies their tangling together in a common module. Scattering and tangling of concerns tend to make programs difficult to evolve, maintain, and reuse. They also tend to complicate the software development process by introducing source control problems when different teams working on different concerns require access to a common module. In the case of components, the tangling of concerns can also introduce configuration management problems.

The philosophy underlying separation of concern technology is to break down a problem into smaller parts. Ideally we want to be able to cleanly separate the different concerns into modules of some kind and explore and develop each in isolation, one at a time. Thereafter, software developers compose these software modules to yield the complete system. Thus, the concept of separation of concerns and the concept of modularity are two sides of a coin – you separate concerns into modules, and each module solves or implements some distinct set

of concerns. Generally, the construction of a framework may adopt the "separation of concern" technology to achieve concern modularization.

We apply the separation of concern technology into our MELC framework for architectural decomposition as well as concern modularization. We show that, just as domain specific classes and groups of these classes with a defined interface, can constitute a domain specific component (distributed computing component), concerns and groups of such concerns can also constitute a component. Our framework is in line with the philosophy underlying separation of concern technology.

Recent research has investigated technologies supporting separation of concerns [48, 49, 50]. Aspect-Oriented Programming (AOP) [49] is one of the most popular aspect technologies today [52]. AOP provides a means to encapsulate the implementation of aspects that are scattered across modules (*crosscut other modules*) [53]. The goal of AOP is to regroup all the elements of a specific concern in a single module. To compose aspects, it is necessary to distribute the elements of the aspect where they are needed. Let's consider the example of a Distributed System [52] in Figure 3.1. The functionality of thread pooling service: the thread pool workers availability is checked, and if a worker is available, it is called upon to perform the task for the server.

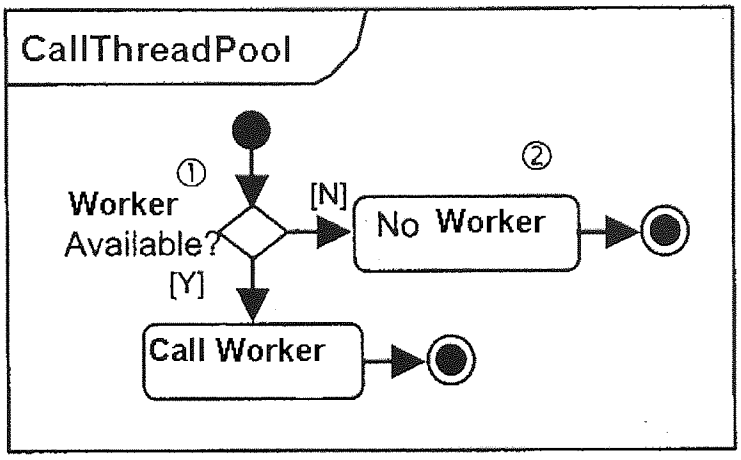

Figure 3.1 Calling a Worker in a Thread Pool in a single module

However, in developing a distributed thread pooling server, a sizable portion of an operation has been found nothing directly related to the functionality of thread pooling, such like start or stop operations in thread pool server, component logging, checking authorization, exception handling, and other such tasks [52] (see Figure 3.2). The redundancy code fragment of Authorization may be found in many other operations and classes.

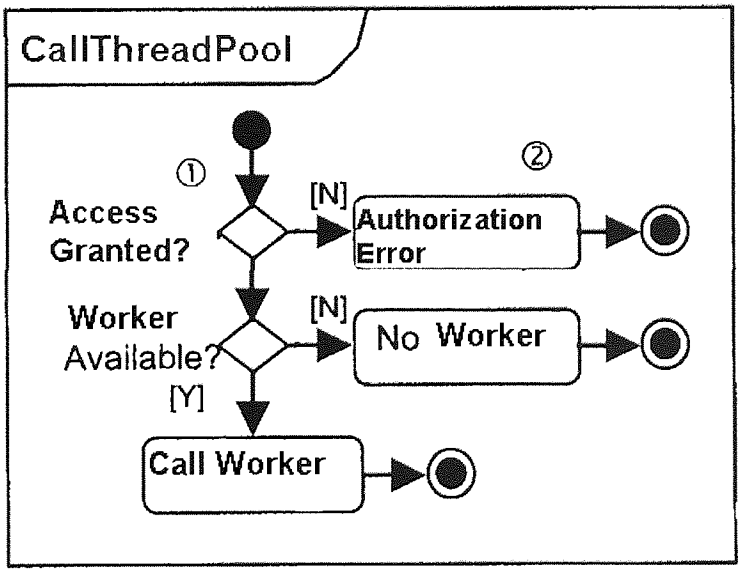

Figure 3.2 Crosscutting Concern – the redundancy code fragment of Authorization found in many other operations and classes

AOP refers to such redundancy as crosscutting concerns because you find these code fragments in many operations and classes in the system – they cut across operations and classes. Crosscutting concerns are not limited to the technical concerns such as authorization and persistence. They include system and application functionality, and you find that a change in functionality often also results in changes in many code fragments in the system.

Current support for AOP from Xerorx PARC is in the form of an aspect language for Java called `AspectJ` [54]. Crosscuts declare points (*join points*) in the code where crosscut actions can take effect. The capability of `AspectJ` from Xerorx PARC to transparently insert modifications (technical concern such as authorization in our example Thread Pooling Service) in a program is extremely powerful.

Our work adopts the separation of concern technology for concern modularization for conceptual modeling. The concept of *crosscutting* and *join points* in aspect-oriented programming (AOP) is adopted to help in designing the framework for managing controls within a component so that the scattering of aspects can be minimized and centralized for maintenance [56].

3.4.2 *Design Patterns and Components*

The separation of concerns (SOC) technique described in the previous section can serve as the decomposition mechanism to identify concerns in terms of components. Localizing all program elements related to a particular concern prevents scattering of these elements across the program code base, making program understanding, evolution, and maintenance easier and less prone to catastrophic disasters. It can also facilitate reuse of the abstract components.

There is a problem that relates directly to the composition of concerns. It is to

find a way to clearly express the semantic resolution in the presence of overlapping concerns. The way of integrating separation of concerns into a component-based software into a more seamless one is to extract the concerns using separation of concerns (SOC) technique, and to realize the concerns themselves into components whose domain specific design patterns are applied in this thesis.

Furthermore, some of the concerns cross-cutting multiple components in a distributed computing environment, such as thread pooling or retransmission, have to be factored out separately into different components. Thread pooling saves the server the work of creating brand new threads for every short-lived task and it also minimizes the overhead associated with getting a thread started. By creating a pool of threads, a single thread from the pool can be recycled over and over for different tasks.

On a system design level in MELC, the components, such as thread pooling or retransmission, are the distributed computing design patterns we applied, and they can be realized and implemented with a programming language and compiled separately. The components integrate with other components to perform thread pooling or retransmission services at the application level in the framework [57].

3.5 Components-Based Framework Development

Component frameworks are the most important step to lift the component software off the ground. 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 component framework establishes environmental conditions for the component instance and regulates the interaction between component instances [44].

We are proposing an evolutionary methodology appropriate for adaptable component based frameworks such as MELC, which evolves from the component-based framework development process [44]. We enhance the process so that the methodology can be used to develop frameworks which employ and provide adaptability for domain specific components instantiated from the patterns. We call it Adaptable Component-Based Framework Development Methodology. The new component-based framework development methodology differs from the traditional waterfall model. As mentioned before, the task of building a system by writing code has been replaced by the process of putting together an application system from reusable software components. The main software development task becomes the creation of the components. Figure 3.3 shows a comparison of methodologies between Waterfall and Adaptable Component-Based Framework Development. Note that the components in MELC framework are domain specific and are the instantiation of distributed computing patterns.

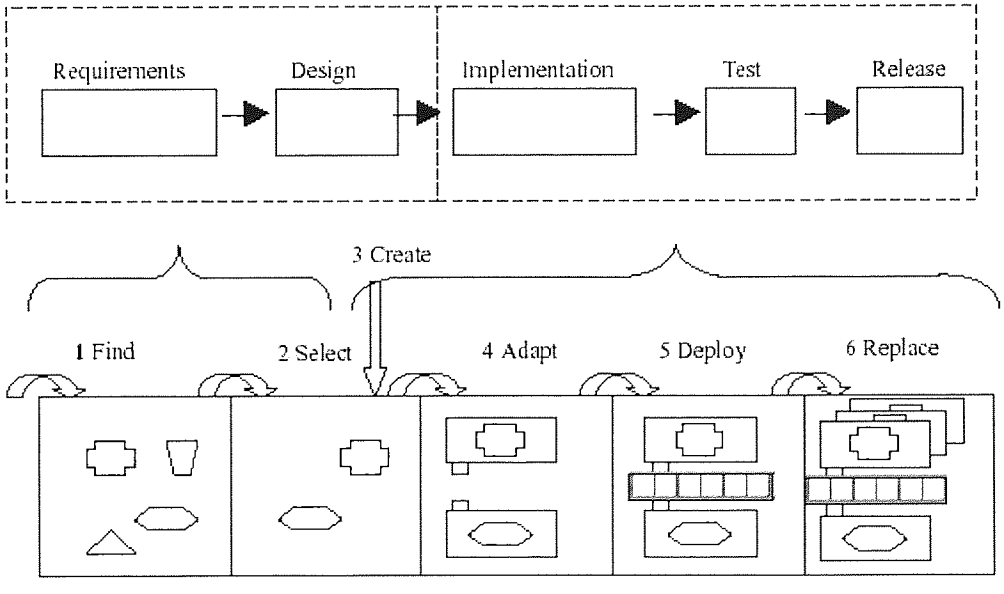

Figure 3.3 A comparison of Waterfall (upper) and
Adaptable Component-Based Framework Development

The six steps in the adaptable component-based development methodology are:

1. Finding the requirements of the components that may be used in the domain specific applications. Component qualification in the methodology is a process of determining the fitness for use of components that are being applied in the system context [58]. In a component-based approach, the high quality and reusable components can be domain specific design patterns.

2. Selection of proper components that fit the requirements of the applications. For a distributed computing application, the proper components in terms of design patterns may be: Object Request Broker, Thread Pooling, Publisher/Subscriber or Mailbox Services.

3. Creation of proprietary components in the framework. The components are instantiated with the selected patterns. The instantiation of a design pattern at the design level in a framework will transform itself to classes which in turn transforms into instances at the lower level, where language independency is changed to language dependency. The mapping is shown in Figure 3.4. Classes generated from patterns will be stored in the form of libraries or packages.

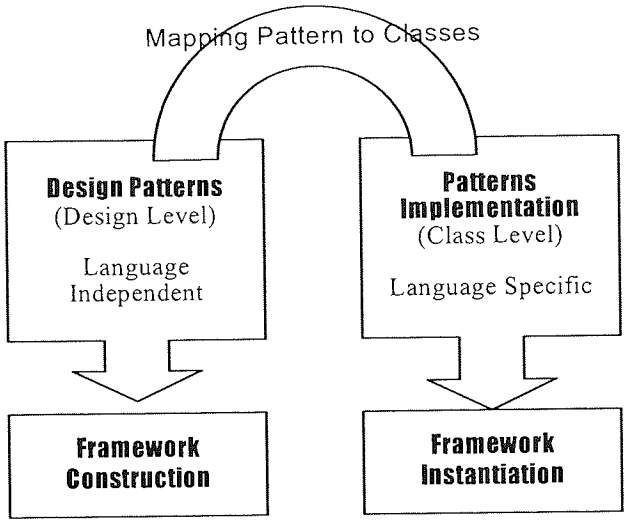

Figure 3.4 Pattern Instantiation in Framework Development

4. Conformation of components to adapt to framework standards. The selected components will be conformed so that the framework establishes environmental conditions for the component instance and regulates the interaction between component instances. The component-based framework is a partial enforcement of architectural principles, by forcing component instances to perform certain tasks through mechanisms under control of the framework.

5. Deployment of the components. The component configuration management can be done with a framework configuration manager. Once the components are deployed, they become part of the services in framework and are managed by configuration manager. The configuration management is the discipline that takes care of component assembly, component configuration and components integration.

6. Replacing old version of the component with new ones. This is also called maintaining the components. There might be bugs that have been fixed. More importantly, the replaced component with new functionality is added as the system evolves and this could happen in a runtime environment.

Chapter 4 Pattern-Oriented Frameworks

4.1 Introduction

The advantage of using design patterns in a framework is that design patterns and frameworks both facilitate reuse by capturing successful software development strategies. A pattern-oriented framework is specified using patterns as building components [6]. Our framework adopts the pattern-oriented approach which uses instances of distributed computing patterns as components in the framework. The pattern-oriented approach offers not only *separation of concerns* but it also adds *extensibility* to the framework. The *separation of concerns* and *extendibility* are the two major properties of our adaptable distributed framework.

There are two categories of patterns: Generic design patterns and Domain specific design patterns. A *generic* design pattern provides a schema for refining the subsystems or components of a software system or the relationship between them. The Strategy, State, and Proxy patterns are examples from this category. A *domain specific* design pattern describes a commonly recurring structure that solves a domain specific problem with a particular context [59]. The heartbeat, subscriber/publisher, and mailbox are examples from the distributed computing pattern category and they can be instantiated as components in a distributed computing framework [14].

The use of patterns in the framework construction improves maintainability. Moreover, using well-established design patterns as components can contribute *separation of concerns* to the constructed framework.

Yacoub et al. [6] first illustrated a pattern-oriented design framework with an

engineering example of a closed-loop control system. In their research, the feedback control framework was presented as a generic architecture based on design patterns. A generic design for such a common problem serves many control system designers in the engineering discipline. It is a small-scale pattern-oriented framework that illustrates the feasibility of the approach for feedback control system.

Pattern-oriented frameworks are white-box frameworks [6]. The framework user has to understand the interface design of the framework in order to adapt the framework in his application. Pattern-oriented frameworks are also domain-specific frameworks [20], as they provide the basic functionality and architecture of applications in a given problem domain.

A pattern-oriented framework introduces two distinct levels, known as the pattern level and the class level.

The pattern level is constructed on well-known or newly invented design patterns and is a design level in a pattern-oriented framework. The pattern level is usually presented in a *Pattern Diagram*. The class level is based on the design of a pattern level and is realized by classes and presented in a *Class Diagram*.

A pattern-oriented framework uses a pattern level as a higher, more abstract design layer above the class level. As described in Chapter 3, the Plotter framework adapts the Template Design Pattern for framework design. The approach of using design patterns in constructing a framework architecture not only makes it independent of any specific programming language, but also improves its design quality and portability.

4.2 Pattern Diagrams

A pattern-oriented framework is specified by patterns as building components and their manner of collaboration [6]. The current support for patterns in UML shows how classes in a class diagram are related to a pattern using a collaboration, which is indicated by a dotted ellipse pointing to participant classes. This is a bottom-up approach, because it marks the classes representing a pattern in the design, but doesn't show it as a design component.

A pattern-oriented framework uses a top-down approach so that the framework designer can start analysis from high-level abstraction in the development process where the framework is partitioned into subsystems and patterns, and then the patterns are exploded into classes. A pattern diagram is used to describe the framework in terms of subsystems, design patterns, and associations.

The elements used in the pattern diagram are shown as below:

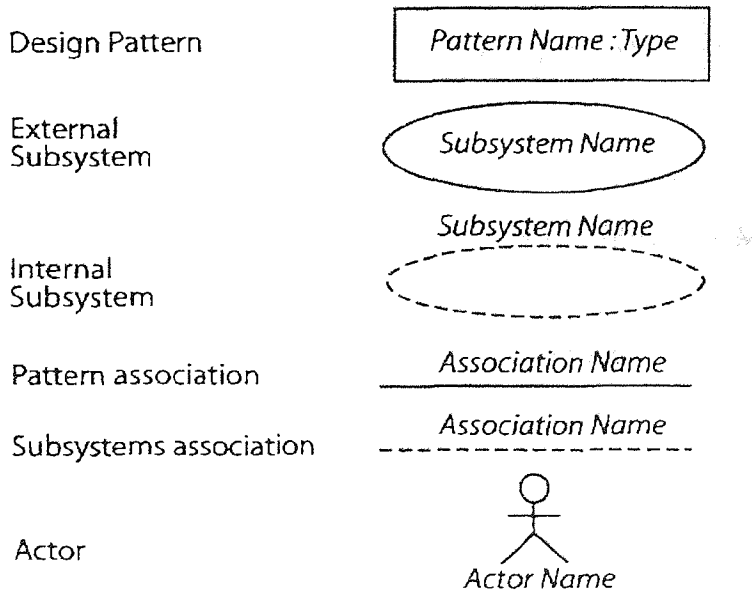

Figure 4.1 Elements in a Pattern Diagram

The terms used in the construction of a framework are defined as:

- Pattern-oriented framework: a construct of subsystems and design patterns that collaborate to describe the generic architecture of a design framework.

- Design Pattern: a design component composed of collaborating classes that are customized to solve a design problem in a particular context.

- Internal subsystem: an independent group of patterns that collaborate to fulfill a set of responsibilities in the framework. The architecture and behavior of a design framework is realized by means of subsystems interacting with each other and with external subsystems. Modularity plays an important role in decomposing a system into subsystems.

- External subsystem: a subsystem that is not part of the framework under development, it can be a part of the real-world environment or other frameworks or applications.

- Actors: external entities interacting with the framework.

- Class associations: relationships between classes, such as aggregation, composition, generalization, and dependency as specified in UML.

- Dependency: in general, implies that the complete functioning of an element requires the presence of another, which exists in the same level of abstraction or realization.

- Pattern associations: relationships between patterns in a pattern diagram. Currently, dependencies are defined as one type of pattern relationship. A pattern dependency indicates a semantic relationship between two patterns;

a situation in which a change to the source pattern may require a change to the target pattern, or a pattern delegates the processing of a certain functions to the other. This relationship is further refined in later design phases to indicate the exact nature of the dependency by translating it into class associations between classes of two communicating patterns.

- Subsystem associations: a general dependency relationship between subsystems in which the change in one subsystem affects the other. A dependency between two internal subsystems implies that one or more dependencies among their patterns exist. The relation is stereotyped further to indicate the exact nature of the dependency, such as calling operation of one pattern to the other, usage of one pattern of the other, etc.

The following example is intended to show how to construct a patter-oriented framework along with a common computerized engineering application, namely *Closed-loop Application System* [6]. The system is subdivided into subsystems based on functionally and behaviorally independent responsibilities. The three subsystems are: *Feedforward Subsystem, FeedBack Subsystem* and *Error Monitoring Subsystem.* The three patterns have been identified and will be used to implement the subsystems, and they are the Strategy [1], Observer [1], and Blackboard Patterns [14]. The way to compile the application requirement and present it as design pattern is called Pattern-Level Instantiation. The resulting diagram is Pattern Diagram. The Pattern Diagram is used to describe the application system in terms of subsystems, design patterns and associations. In the Figure 4.2, each subsystem is represented in the form of design patterns.

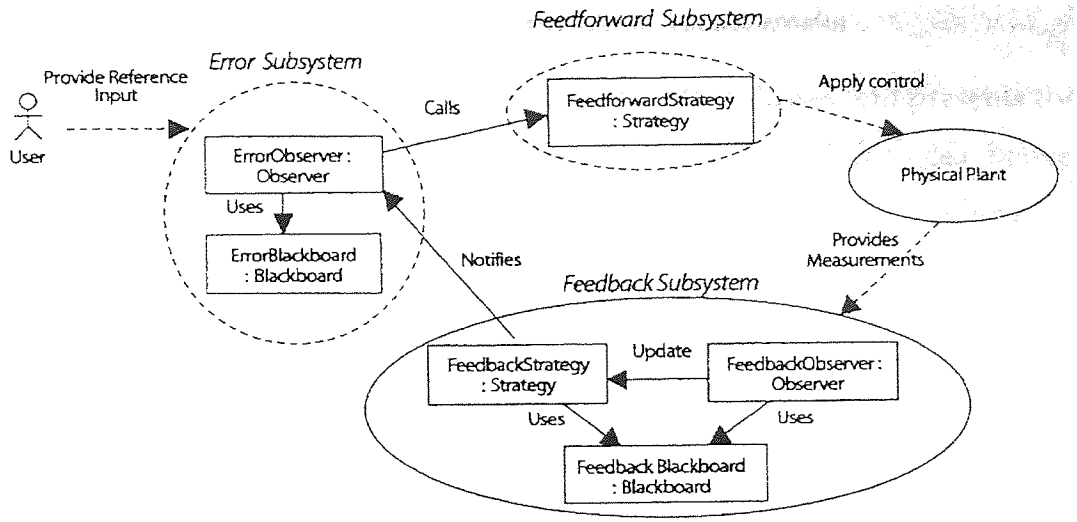

Figure 4.2 Pattern Instantiation of Closed-Loop Application System [6]

The associations referred to in Figure 4.2 represent relationships between patterns and relationships between subsystems. An association between patterns is represented by a solid line joining the patterns and is labeled with an association name. A relationship between subsystems is represented by a dotted line joining the subsystems and is labeled with the association name. Arrowheads in an association are used to show the association direction when applicable.

4.3 Pattern-Oriented Frameworks with Other Frameworks

Most of the work refers to frameworks as collaborative classes and presents them as class participants whenever patterns are used. The differences between pattern-oriented frameworks and other frameworks are identified as follows.

- *Pattern-Oriented Frameworks have Pattern Level views*

Beck and Johnson [60] show how to use patterns to document a framework, and show how patterns can be used to derive the framework HotDraw. It eases understanding of the final system, but does not explicitly discuss how the patterns interact and collaborate together. However, a pattern-oriented framework not only documents the framework, but also it can be used to design the frameworks defined by the architecture of communicating design patterns in *pattern-level view*, which describes the interaction and collaboration of the patterns. The patterns are further extended to classes and objects in a *class-level view*. Framework construction in terms of design patterns adds to the reusability and descriptive capability of the design framework because patterns can coexist and may depend on one another. Moreover, using well-established design patterns as components can contribute to the design quality of the constructed framework. Pattern-Oriented Frameworks use Pattern Level views to show how patterns interact and collaborate.

- *Pattern-Oriented Frameworks are White-Box Frameworks*

Johnson [61] describes how patterns can be thought of as micro-architecture elements. Hence a single framework contains many patterns. Pattern-oriented frameworks are closely related to his definition. A pattern-oriented framework introduces a pattern-level architecture description of a framework in terms of a pattern diagram that we perceive as an approach to solve the design of components, and addresses reuse at the component-based design level, which differs from code fragment reuse.

- *Pattern-Oriented Framework adopts Top-down design approach*

Castellani & Liao [62] and Jia [13] propose an application development process that focuses on the reuse of OO application design. Their work presents processes that can be used by a system designer to create generic applications and reuse them in other application designs in the same problem domain. Their approach starts with an existing application and then abstracts design macro-components through application specifics. However, the authors use a general definition of macro-components (frameworks or patterns) that allows any group of related classes to be considered a pattern. This property is obviously true for a package or library in programming, but it seems too weak for a definition of a pattern. An application, in the context of their development process, is split into macro-components, which are filled later on with classes. A pattern-oriented framework adopts the top-down design approach. It tackles the problems found at the design level from the perspective of constructing design frameworks with appropriate patterns in the related problem domain.

The top-down design approach for building a pattern-oriented framework starts from an architectural design based on a pattern diagram, proceeds with expansion of patterns into their predefined class diagrams, and further refines the class diagram through reduction and grouping phases to reach a final class diagram of the framework.

- *Pattern-Oriented Framework uses patterns to build system architecture*

Odenthal and Quibeldey-Cirkel [63] describe the benefits of using design patterns as an intermediate level of systems description between the analysis and design levels. They address two important issues: deploying patterns in designs, including the identification of candidate patterns, and the addition of a documentation level based on patterns.

The pattern level in a pattern-oriented framework reduces the descriptive complexity by covering the design with pattern instances. Identifying candidate patterns is closely related to pattern-oriented framework construction; however, the authors do not show how the selected patterns are put together to construct the overall system. The patterns will be instantiated and integrated in a framework. Their collaboration must be documented and, lately, they can be implemented accordingly. In the construction of a pattern-oriented framework, candidate design patterns are identified, thereby describing the overall system in terms of patterns.

- *Pattern-Oriented Framework uses patterns as building blocks*

Schmid [64] discusses the architecture for manufacturing systems. He starts from an existing OO analysis and targets a more general and flexible architecture for automated manufacturing systems like assembly lines. Although successive transformation steps from requirements analysis to system design are guided in identifying the pattern candidates to create an overall architecture, patterns are not used as design building blocks for the construction of the framework. Pattern-Oriented Framework uses patterns as building blocks. Patterns are considered and put in place in architecture design.

4.4 Pattern-Oriented Framework Development

The development of a framework is like the development of most reusable software. It starts with domain analysis, which collects a number of same domain specific examples. One of the most common observations about framework design is that it takes place over a series of iterations. Versions of the framework are designed to implement the examples which initially create a white-box framework. Each example that is considered makes the framework more general and reusable. When it is complete, the framework can be used to build applications. Experience leads to improvements in the framework.

Figure 4.3 [6] shows the first step in a requirements analysis which is the analysis phase. This step is necessary because understanding the system can pose a problem even for experts unless the domain is well-defined. Hidden mistakes in a domain analysis can be discovered when a system analysis is being undertaken.

A framework is structured into subsystems in Figure 4.3, which contain objects that are functionally dependent on each other. The goal is for objects with high coupling among each other to be in the same subsystem, and objects that are weakly coupled to be in different subsystems. A subsystem can be considered a composite or aggregate object composed of individual objects. Packages can be used to represent subsystems. Thus, one package representing the whole framework may be decomposed into subsystems, where the subsystems are shown as nested packages within the framework package.

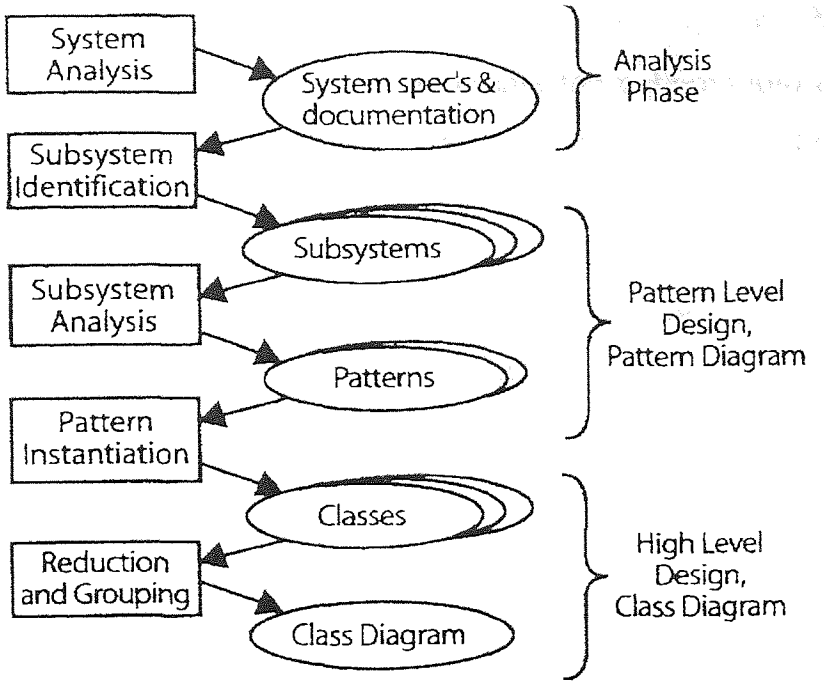

Figure 4.3 Development Process in Pattern-Oriented Framework

Figure 4.3 illustrates the steps involved in the development of a pattern-oriented framework. The rectangular boxes show the activities, and the ellipses show the inputs and outputs of each activity. The development process involves the construction of a pattern diagram which is eventually reduced to a final class diagram.

With more than 100 design patterns in the catalogs for general purposes and domain specific like distributed computing applications, it might be hard to find appropriate design patterns for a particular design problem. Several different approaches to selecting a design pattern have been suggested by Gamma and el. [1] and Grand[14]:

- Scan intents of design patterns. Read through each pattern's intent to find one or more that sound relevant to our problem.

- Study how patterns interrelate. Instantiate the patterns into classes and study the interrelationships between them. The contradiction issue identified between patterns could be analyzed and resolved at the stage of implementation at the class level by software developers. For example, although the single and multiple thread ThreadPool patterns contradict each other in functionality, they can both exist in the framework at the same time and can be used by software developers at different places in an application created using the framework.

- Consider how design patterns solve design problems. Find appropriate objects, determine object granularity, specify object interfaces, and ways in which design patterns solve design problems.

4.5 Summary

This chapter presented the pattern-oriented approach to design frameworks with generic patterns and briefly outlined the elements in pattern diagrams and the flow for framework development process. Employing a pattern-oriented approach has lots of advantages: it tackles the problems found at the design level from the perspective of constructing design frameworks with appropriate patterns in the related problem domain, and it addresses portability at pattern level.

The pattern-oriented approach for the framework construction is a top-down design. The pattern level in a pattern-oriented framework is used as a higher design layer than class diagrams. Framework users can start with the pattern-oriented framework at the pattern level, which reduces development time and effort to start up from architecture and class diagrams.

Our framework adopts a pattern-oriented approach. The conceptual modeling in designing a framework in terms of design patterns adds extensibility and maintainability to our framework. Moreover, using domain specific design patterns (i.e., distributed computing patterns) as components not only contributes to the design quality of the constructed framework, but it also provides separation of concerns in framework. Note that *Separation of Concern* and *Extensibility* are two key properties required in an adaptable framework architecture identified in the Chapter 2. The related works in *adaptability* for components in system evolution at runtime will be illustrated and addressed in the next two chapters.

Chapter 5 Theoretical work on Adaptability in Meta Architecture

Adaptability is an increasingly important requirement of software systems. An adaptable architecture is intended to provide flexibility by providing an architecture that allows requirement changes to be performed and immediately reflected at run time. Minimizing downtime is critical in distributed computing environments, such as web services, telecommunication, banking systems, military missions systems, and life support systems. It is particularly important that a reflective architecture should be able to change an object's behavior to fulfill user requirements to adapt at run time. In mission critical cases, the framework architecture can not be taken offline for maintenance. There is a strong argument to support the need for reflection in a framework architecture in order to provide adaptability in object behavior at runtime. *Adaptability* is an indispensable and key property for the framework which we will present in this thesis.

In Chapter 4, we presented a pattern-oriented approach to designing frameworks based on design patterns. The resulting frameworks are called pattern-oriented frameworks. However, the pattern-oriented approach has some drawbacks related to adaptability:

- The patterns chosen in the framework may be good for now but not for tomorrow. Replacing a pattern in an implemented pattern-oriented framework may be costly.

- The methodology is not applicable to all application-specific frameworks in general. For example, loop-back systems are common engineering applications, and they are commonly described in block diagrams. Most

distributed business applications may not have a clear system architecture at the start. Always, errors can be made when choosing appropriate, good-fitting patterns.

- Users have to know the class-level details of the pattern-oriented frameworks before maintaining them. This makes pattern-oriented frameworks less user-friendly for framework developers because they seem to be designed only for computer programmers.

The adaptable framework architecture is designed to dynamically adapt to new user requirements by replacing software components at run time while the rest of the system is still running. To add such adaptability to a framework requires a *Reflective Architecture* which we call a *Meta Architecture*. The terms, Reflective Architecture and Meta Architecture, go hand in hand and are interchangeable [2].

Adding a meta architecture to a pattern-oriented framework helps to resolve the problems of adaptability encountered in pattern-oriented frameworks and keep the advantages, separation of concerns and extensibility, provided by the pattern-oriented approach.

The following sections discuss the use of *Reflection* in Meta Architecture and briefly discuss *adaptability* in recent research work on the reflective models. At the end of the chapter, we provide a table to summarize the features, which compares each of the presented models against the common assessment attributes [100] required by the academe and industry.

5.1 Reflection in Meta Architectures

The adaptability of a framework is determined by how well the framework faces software evolution because of the newly invented techniques or the improvement of existing features in distributed computing. Meta Architecture can provide adaptability and also be flexible enough to meet a wide range of requirements on demand. Most domain-oriented frameworks, unfortunately, are monolithic and provide a fixed and limited set of capabilities. It is typical to take the scrape-and-build approach for a given requirement, where the software is rewritten from scratch because that approach is often determined to be more economic. On the other hand, a dynamically adaptable meta-architecture based framework is an attractive alternative for extensive and intrusive changes in distributed computing.

Suzuki and Yamamoto [4] pioneered the application of meta architecture to design a web server which supports the dynamic adaptation of flexible design decisions in the web server design space. Their framework is intended for web servers only, with no consideration of a generic component-based approach. It does not provide a Configuration Management utility for managing the components nor does it support dynamic changes in the runtime environment. However, it demonstrates the possibility of applying a meta architecture to attack the problem of framework adaptation.

A meta architecture differentiates between *meta objects* and *base objects*. A meta object is an object that contains information about the internal structure and/or behavior of one or more base objects. A base object has application logic and behavior of method invocations sent by the clients. Its implementation builds on the meta object. Changes to information kept in the meta object affect subsequent base object behavior. Meta objects can be used to track and control certain aspects, such as structure and/or behavior, of base objects.

Meta objects are grouped into a *meta-space* which we will call the *meta level,* and base objects are grouped into a *base level.* Figure 5.1 below illustrates the relationships between the meta level and the base level.

An important aspect that determines the applicability and performance of a meta architecture is the way the connection between meta level code and base level code is established and how control and information flows between the two levels.

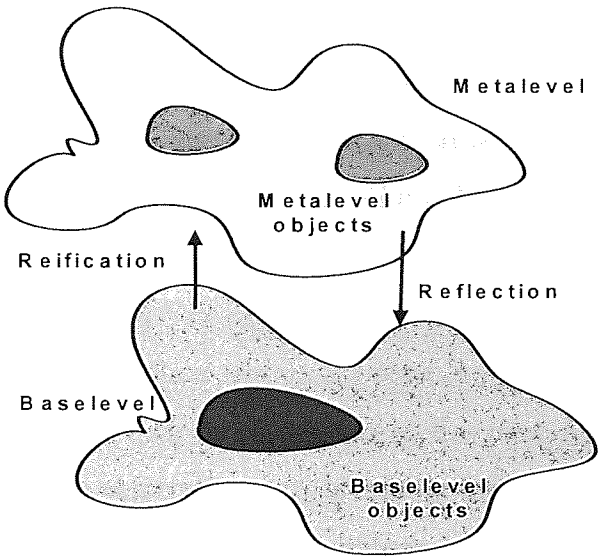

Figure 5.1 Meta level and Base level in a typical meta architecture

Figure 5.1 shows that a meta architecture has two functions in a reflective framework, *Reflection* and *Reification.* They are different in the way the two levels are accessed.

Reflection is the ability of a meta object to manipulate data that represents the state of a base object and to adjust itself to changing requirements. Another way to describe reflection is by means of Causal Connection [74]. Causal connection is related to the connection between Description and Described Object. In a

reflective system employing a meta architecture, each meta object represents or describes certain aspects of the implementation of a base object [4]. Changes in the meta objects (descriptions) result in changes to how the base objects (described objects) are implemented. *Reflection* means that a system can manipulate a causally connected description of itself [65].

The causal connection is established when the base object (described object) receives the message which is interpreted at the meta level. The meta level decides what to do with the message. In other words, the meta level can describe what the message is supposed to do. Causal connection implies that changes to the description have an immediate effect on the described object. A good example to demonstrate *reflection* is the relationship between a subject and its observers. Once the subjects state changes, all its observers will immediately be notified.

In contrast to reflection, *reification* makes it possible for meta data, hidden from the application developer [4], to be available at the base level process. A meta architecture allows a base object, through accessing meta objects, to reason about its own execution state. The relationship between a subject and its observers can also serve as an example to demonstrate *reification*. Each observer can alter its behavior based on the current state of the subject to which the observer corresponds.

Meta objects may be thought of as objects, which logically belong to the underlying run-time system. For example, a meta object might control the message lookup schema that maps incoming messages to operations in the base object. Another meta object may modify how values are read from memory. This yields a potentially customizable run-time system within a framework [65].

A Meta Architecture supports a layered structure within its meta level. Meta

objects can be associated with their meta objects. Each meta object may be defined separately or be composed of other meta objects. Thus the latter provides a layered structure supporting reuse and incremental construction.

To supervise the execution of a base object, it has to be explicitly reified into the corresponding meta level. A set of interfaces with which a base-level object accesses its meta level constitutes Meta-Object Protocols.

5.2 An Example of The Meta-Architecture Approach - A Simple Drawing Pad

A simple drawing pad application [13] is used to demonstrate the reification and reflection in meta level in meta architecture. A screenshot of the simple drawing pad is shown in Figure 5.2. The drawing pad supports multiple tools which are scribbling tool, eraser tool and line tool.

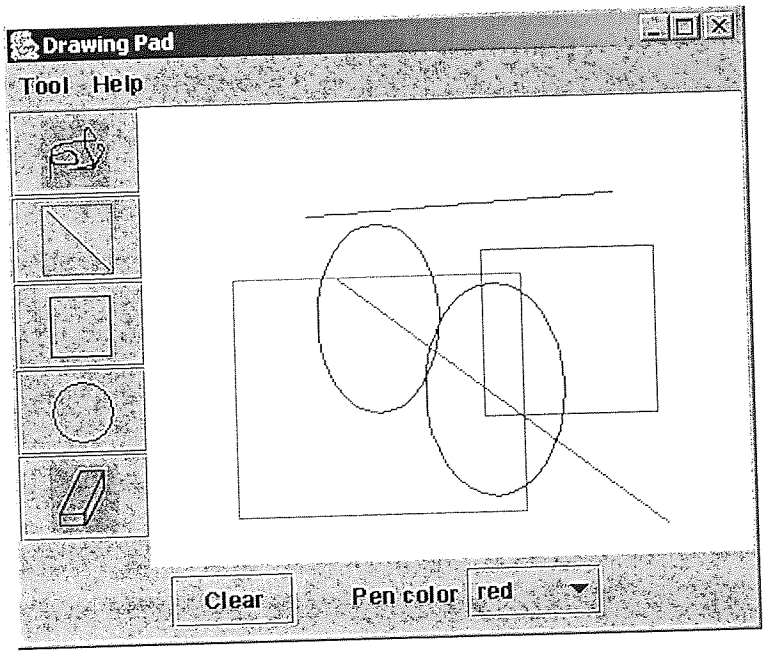

Figure 5.2 Drawing Pad

Meta Level Objects

The structure of the meta level objects is shown in Figure 5.3. The classes are summarized in the following table.

Class	*Description*
ScribbleMeta	The initial controller class
ScribbleCanvas	The double-buffered drawing canvas
ScribbleCanvasListener	The event listener that listens to mouse events in the drawing canvas

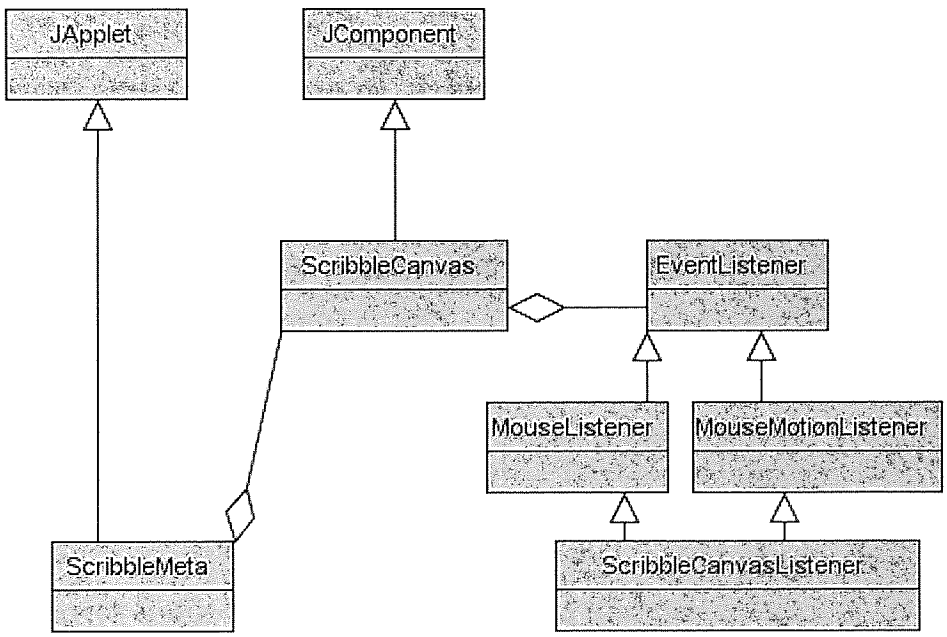

Figure 5.3 Structure of meta classes at meta level

`ScribbleMeta` is a meta level class because it contains information about the internal structure and behavior of base level object. It provides drawing canvas and mouse event listeners for the drawing pad. We can create a base level class, `DrawingPad`, by extending the meta level class `ScribbleMeta`.

The fields and methods of the `ScribbleMeta` class are summarized in the following table.

Member	*Description*
canvas	Drawing canvas
listener	Mouse event listener of the drawing canvas
isApplet	Whether the scribe pad is invoked as an applet (It could be an application)
makeCanvas()	Factory method that creates the drawing canvas
makeCanvasListener()	Factory method that creates the mouse event listener
main()	Public method for the scribble pad to be invoked as an application

As `ScribbleMeta` is developed at the meta level it provides only minimal functionality of a scribble pad. The canvas instance has only a blank canvas for drawing, which does not have either a tool bar or menu bar on it. The listener instance created by can only listen to events like `mousePressed`, `mouseDragged` and `mouseReleased`.

In this application, adaptability at the meta level is demonstrated by dynamically adding a new meta level class on demand. `Scribble2Meta` extends and enhances `ScribbleMeta` to create a control panel that contains a clear button to clear the canvas and a choice control to choose a new pen color.

`Scribble2Meta` contains an extra method that creates the control panel containing a Clear button to clear the canvas and a choice control to choose a new pen color. The screen shot of the enhanced scribble pad, `Scribble2Meta`, is shown in the Figure 5.4, and the structure of the enhanced scribble pad is shown in Figure 5.5.

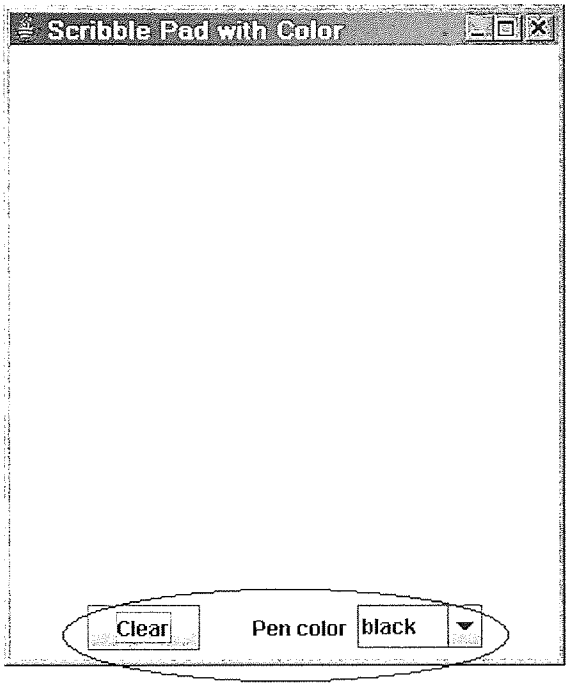

Figure 5.4 Scribble pad screen of meta object Scribble2Meta

We can say that the meta level class in meta architecture allows unlimited adaptability of meta level classes. `Scribble2Meta` is an enhanced version of `ScribbleMeta`. Most of the current classes in meta level in meta architecture can be reused for future upgrades. However, they are limited to the *class level* in object-oriented software development.

When there are any changes in meta level classes because of system evolution, the classes are required to recompile and have all their objects instantiation to stop for system upgrades in system server. We know that it is unacceptable solution for most mission critical systems such as E-banking and Life Support systems. This yields a potentially customizable run-time issue within a framework [65].

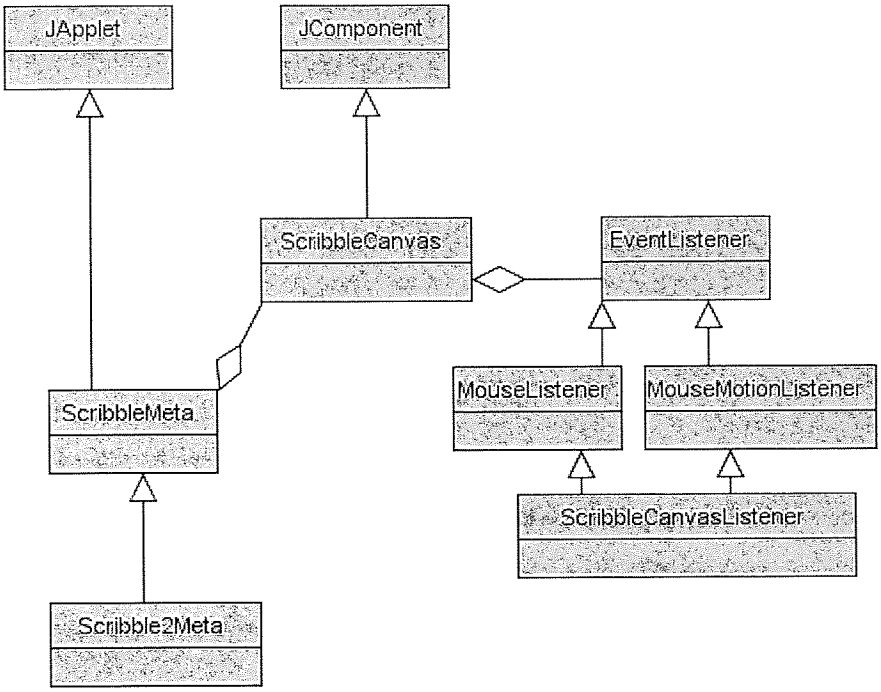

Figure 5.5 `Scribble2Meta` extends `ScribbleMeta` at the meta level

To resolve this issue, the Meta-Level Component-based Framework, MELC, developed in this thesis, will show the solution of applying the meta architecture to provide adaptability at the object level in the runtime environment (*not* class level). Such a reflective framework architecture is flexible enough to meet a wide range of requirements on demand; not only will the framework allow the insertion of meta level objects; it should provide opportunities for the replacement of the established meta level objects after instantiation in the framework server.

We believe software systems require an adaptable framework architecture to be designed that will dynamically adapt to new user requirements by upgrading their software components at runtime, leaving the rest of the system intact. With this drawing pad application, we try to emphasize that MELC - a dynamically adaptable meta architecture based framework in the run time environment is an attractive alternative for extensive and intrusive changes, which provides adaptability for objects at the meta level in the meta architecture.

Base Level Objects

The drawing pad supports multiple tools which are scribbling tool, eraser tool and line tool. The fields and methods of the `DrawingPad` – a base object are summarized in the following table. The structure of the drawing pad's base object is illustrated in the class diagram shown in Figure 5.6.

Member	*Description*
`CurrentTool`	Current tool being selected
`makeToolBar()`	Auxiliary method that creates the tool bar
`makeMenuBar()`	Auxiliary method that creates the menu bar
`makeCanvasListener()`	Factory method that creates the mouse event listener

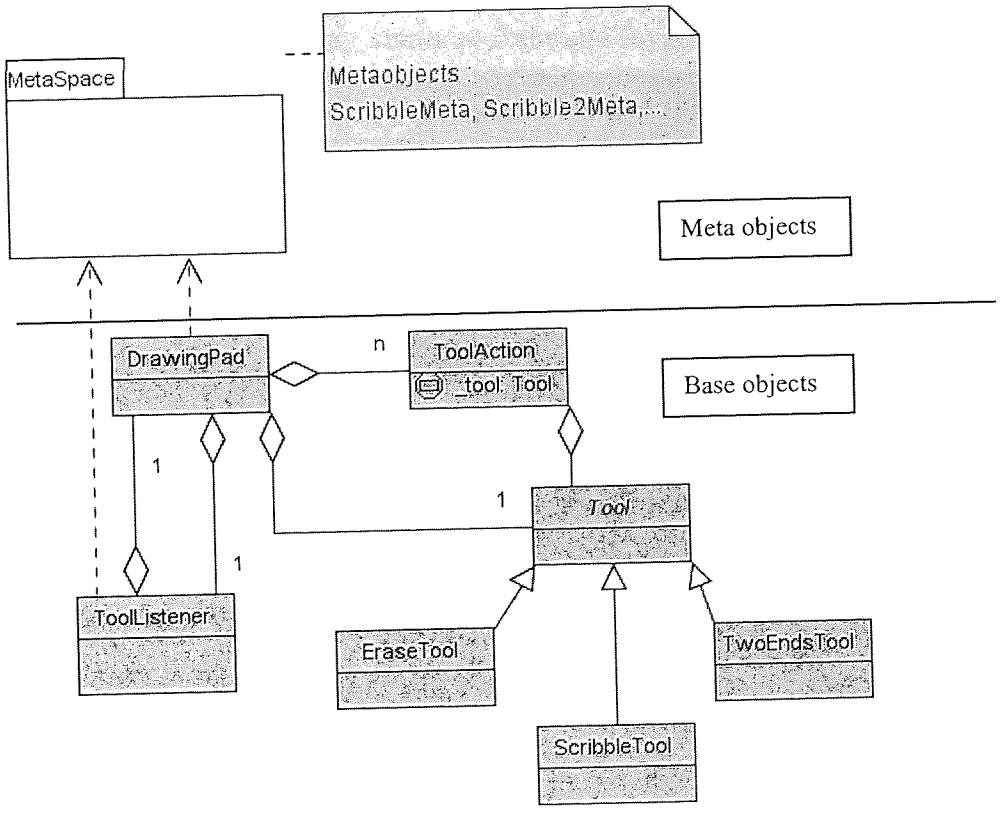

Figure 5.6 Structure of Drawing Pad – A Base Object

The drawing pad as shown in Figure 5.2 has a tool bar at the left, which is used

to select the current tool. Each button represents a different tool.

The base object `DrawingPad` extends the meta object `Scribble2Meta`. The `DrawingPad` has two methods. One method is used to construct the tool bar at the left, which creates a button with image and an action for each tool. Another method is used to construct the menu bar at the top, which creates a menu item for each tool.

In the `DrawingPad` class, the creation and registration of the `ToolListener` object can be used to demonstrate the Reflection and Reification in meta architecture model. As we have mentioned, reflection means changing the states in meta level will result in changing states in base level. In contrast to reflection, reification is the process of making states at the meta level to be accessible to base level. They are two different directions for processing.

Reification in Drawing Pad

The control is from the meta level, which is not normally available in the base level in a programming environment, and is hidden from users. `DrawingPad` extends `Scribble2Meta` which in turn extends `ScribbleMeta`. *Reification* in meta architecture happens here. During the instantiation of the base class `DrawingPad`, the construction of `DrawingPad` will instantiate meta object `Scribble2Meta`, and then also will instantiate meta object `ScribbleMeta` dynamically.

The constructor method in the base object reifies the meta object, `Scribble2Meta`, in meta level. After the process of *reification*, the meta object `Scribble2Meta` will be accessed by the base object `DrawingPad`.

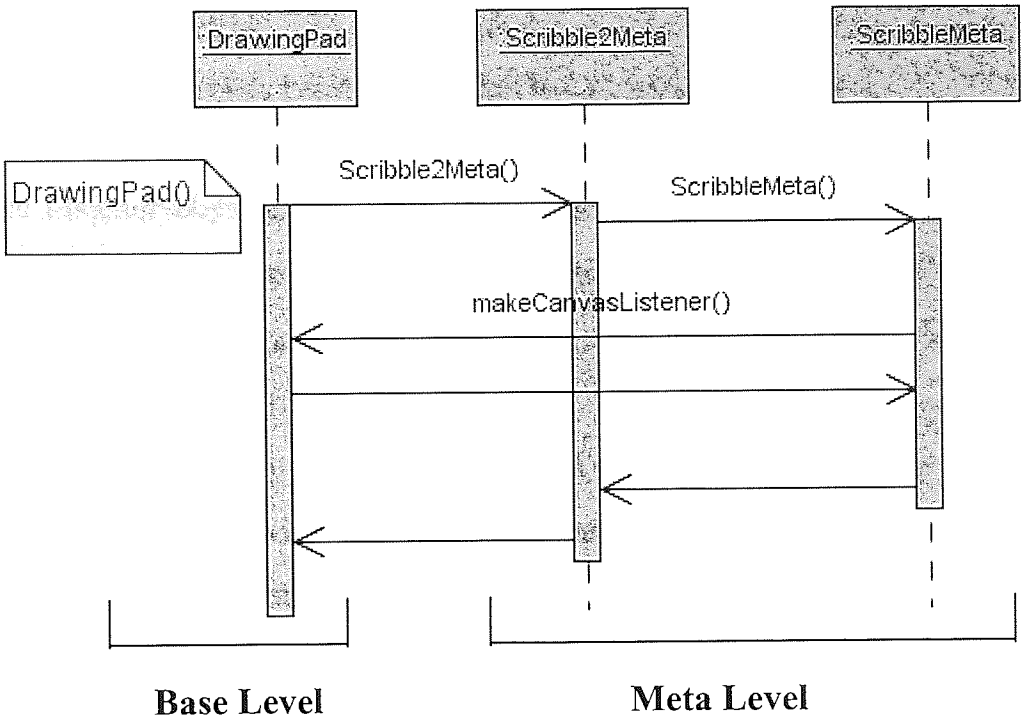

Figure 5.7 Sequence diagram of the constructor method in `DrawingPad`

At the meta level, the constructor of `ScribbleMeta` calls the method `makeCanvasListener` to create a listener object, `CanvasListener`. The Listener object is one of the data members belonging to the meta object `ScribbleMeta`. However, the base object `DrawingPad` overrides the method `makeCanvasListener`. The meta object `ScribbleMeta` will call the method `makeCanvasListener` in base object `DrawingPad` to create the listener object.

The sequence diagram of invocations of constructors of the superclasses is illustrated in Figure 5.7. In the implementation, the constructors of the superclasses `ScribbleMeta` and `Scribble2Meta` are invoked by the constructor of `DrawingPad` class. The meta object `ScribbleMeta` will call the base object `DrawingPad` to create listener object as we have described earlier.

Reflection in Drawing Pad

Reflection is the ability of a meta object to manipulate as data that represents the state of the base object and adjust itself to changing requirements.

The Listener object is one of the data members belonging to the meta object `ScribbleMeta` and represents the state of the base object `DrawingPad`. *Reflection* in meta architecture happens here. Changing the listener will affect the behavior of the system when handling events encountered on canvas.

The listener object will be different depending on the class that constructed it, `ScribbleMeta` or its subclass `DrawingPad`. The listener object could be a simple `EventListener` as provided by `ScribbleMeta` or it could be an enhanced listener that also listens for events related to the tool bar that `DrawingPad` class has created. Creating an enhanced listener requires the `DrawingPad` class to override the `makeCanvasListener` method.

In the base level, reflection allows the base object `DrawingPad` to have its own listener and, consequently to have its own state and behavior during execution. The sequence of invocation involved in using the factory method is illustrated in Figure 5.7.

In the constructor of the superclass `ScribbleMeta`, the `makeCanvasListener` method creates an instance of the `EventListener` object and registers it as the mouse listener of the drawing canvas.

Method of class `ScribbleMeta`: `makeCanvasListener`

```
protected EventListener makeCanvasListener(ScribbleCanvas canvas)
{
  return new ScribbleCanvasListener(canvas);
}
```

The relevant methods of the `ScribbleCanvasListener` class are summarized in the following table:

Methods	*Description*
`mousePressed`	Handles the mouse button pressed event
`mouseReleased`	Handles the mouse button released event
`mouseDragged`	Handles the mouse dragged event

However, the method `makeCanvasListener()` in subclass `DrawingPad` overrides the method in `ScribbleMeta` and creates a different instance of an `EventListener` object. The mouse event listener associated with the drawing canvas simply delegates the handling of mouse button presses and releases and mouse dragging to the current tool.

Method of class `DrawingPad`: `makeCanvasListener`

```
protected EventListener makeCanvasListener(ScribbleCanvas canvas)
{
  return new ToolListener(this, canvas);
}
```

As mentioned before, the `Listener` object is one of the data members belonging to the meta object `ScribbleMeta` and represents the state of the base object `DrawingPad`. *Reflection* in meta architecture happens here. Changing the listener in meta level will affect the behavior of the base object `DrawingPad` when handling events encountered on canvas.

We can say that the creation and registration of the ToolListener object in the drawing pad application demonstrate the Reflection and Reification in a meta architecture model. However, the implementation of the drawing pad application is limited to the non-distributed system environment.

The Meta-Level Component-based Framework, MELC, in this thesis, aims at providing adaptability for distributed system environment. It embraces lightweight technology such as Object Request Broker (ORB) which transparently hides distribution and other non-functional concerns from software developers. Thus, our MELC is a component-based framework which is enabled to make distributed systems configurable, flexible, and adaptable [85].

5.3 Related Works in Reflective Models

Recent work has concentrated on reflective architecture for adaptation concerns. They have proposed quite a few theoretical approaches to build reflective systems to meet the changes of objects for adaptation during runtime. Their approaches focus on the dependencies between objects [66, 67], the interfaces of objects [68], the interactions between objects [69, 70], or the changes of the behavior of objects [71]. However, the approaches still can not come up a feasible framework architecture to meet the demand for runtime adaptation for system evolution. We summarize their works in this section.

5.3.1 Object Dependencies in Reflection

Dependencies, which specify relations between objects, can be changed during run time to achieve adaptation. Lunau [67] has proposed a reflective architecture for process control applications that handles dependencies by using a system meta object. In process control applications, several independent aspects of behavior need to be monitored simultaneously, such as fault detection, logging of values, and reconfiguration of the process. The monitoring can be performed by having a large set of meta objects which contain the different aspects to be monitored. The system meta object administrates the composed meta objects, and invokes them in turn. The architecture is shown in Figure 5.8.

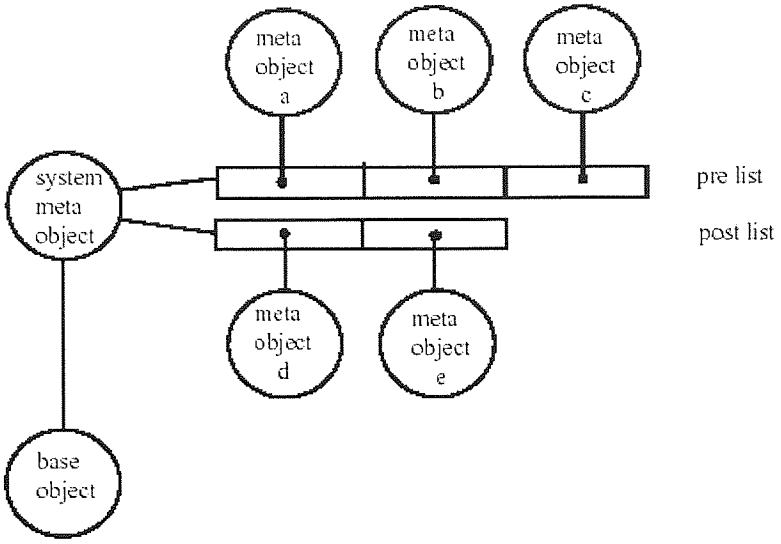

Figure 5.8 Composition of Meta Objects in Meta Architecture

The system meta object handles two lists. On the *pre list* are all meta objects that should be notified before the operation in the base object is performed in the base object. The *post list* contains the meta objects that need to be notified after the performance of the operation in the base object. The meta objects are supposed to be dynamically added and removed from the lists for system evolution. Unfortunately, it is very difficult to replace the meta objects in the lists due to cohesive dependency between system meta object and its meta objects in the lists at run time environment. We consider an adaptable framework should provide *adaptability* that does not require meta objects (functions) to possess cohesive dependency so that they can evolve independently.

5.3.2 *Interfaces Realization for Reflection*

Walker et al. [68] have proposed adjusting the parameters passed to an object during run time to implement contextual reflection. Contextual reflection consists of a meta level which intercepts method invocations and inserts the parameters that the object expects, or performs a mapping between interfaces. The meta level can inspect objects and holds the history of method calls to objects.

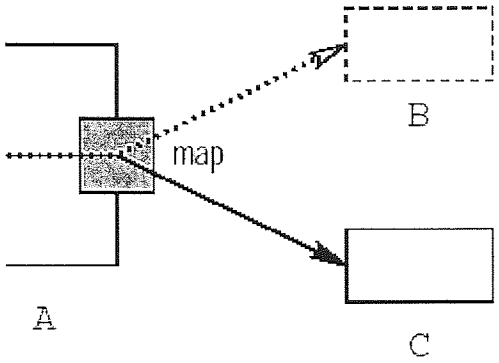

Figure 5.9 Mapping performed at the Meta Level

The architecture is shown in Figure 5.9. An outgoing message is intercepted at the boundary of mapping component A in the meta level. The message contents can be manipulated and the message rerouted to a new recipient C, all based on the mapping at the meta level. The mapping component A plays the role of an *adapter* (see adapter design pattern) in the architecture, which depends on the predetermined path (history) of the method calls to the object. In this approach, the intercepted messages (methods) are not aware of the adaptation of their interfaces.

The *adapter* in this approach provides a fixed mapping between interfaces. The mapping is system-determined and cannot be changed during run time. Such dynamic adaptation is limited to the anticipated environment only. According to our requirements for an adaptable framework mentioned in Chapter 2, the framework architecture should have *extensibility* required to support both adaptation in the anticipated environment and also to handle unanticipated adaptation during run time.

5.3.3 *Objects Interactions for Reflection*

Lead [69] proposed the control flow between components, to controls the interaction between components. The control flow can be changed during run time depending on the computation described by *strategy objects* which intercept method calls and use the information on the actual environment of the application to control the interaction between components. Lemos et al. [70] have proposed another approach, a reflective software architecture which changes a set of interactions between objects and is called cooperation during run time. It uses *cooperation managers,* implemented as meta objects, to select the way objects interact with each other during run time. Both the reflective approaches [69,70] achieve separation of objects and the adaptation of their interactions. However, meta objects in these reflective approaches cannot be replaced at runtime [70]. Their reflective approaches fail to meet the requirements of an adaptable framework in the aspect of *adaptability* at runtime.

5.3.4 *Roles Management for Reflection*

Tramontana [71] has proposed a behavioral reflective architecture that is able to adapt its components by addressing the problem of changing the behavior of objects during run time using *roles*. Reflective systems separate functionality and adaptation strategy. Roles are modeling abstractions able to capture specific views of the object. Let's consider the example of a Hotel Reservation System [52]. Making a room reservation: first check the room availability, make the room reservation, and update the room information. The role of a customer is different from that of a hotel supervisor. The role of a customer is to request a reservation. The hotel supervisor role has privileges necessary to execute all functions related to making room reservations. Some rules are embedded in the role objects, which make the behaviors of customer object and hotel supervisor perform differently.

A *role manager* expresses the adaptation strategy of its associated object by some rules that establish when the object can change its behavior using roles. A

cooperation manager is used to coordinate the request for accessing a group of objects and participating roles. The relationship between objects and roles are associated with performing component functions at the meta level.

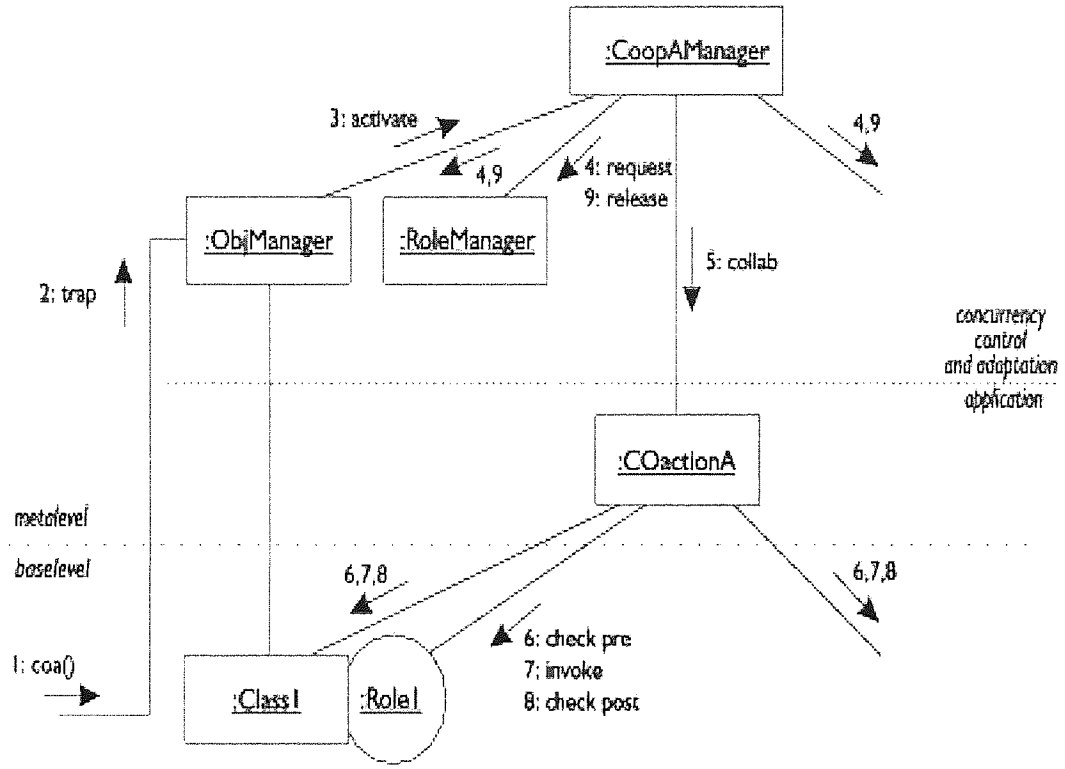

Figure 5.10 Roles Management for Reflective Software Architecture [71]

The architecture and the sequence of invocations for changing object behavior using roles are shown in Figure 5.10. The invocation (1) of the method of an object at base level is trapped to its object manager (2) which gives control to a cooperation manager (3). The cooperation manager selects, during run time, the role object to participate in the associated cooperation. If the rules for adaptation are satisfied, the role manager (4) attaches the role object to the base object at base level. When objects and roles at the base level have been acquired, the cooperation manager activates the cooperation (5) which coordinates and operates on the selected objects and roles (6,7,8). When the invocation has been

completed, the objects and roles are released by the cooperation manager (9).

However, the role objects in the relation cannot be easily changed or evolved dynamically at runtime environment when they are associated with the object in the relation. The relation of the objects and their roles are closely coupled and are firmly bound together in the run time environment and they cannot evolve independently. Tramontana's approach fails to meet the property – *adaptability* of an adaptable framework.

5.4 Summary of Reflective Models

Reflection is a technique that provides the basic mechanism to build an adaptable software model for managing system evolution. It combines aspects in *adaptability* for system development to aid the evolution of running systems.

In Table 5.1, we categorize the related work on reflective models according to the assessment attributes [100]: meta object co-ordination, meta object integration, meta object adaptation, and communication linkage between base level and meta level. We also evaluate each reflective model included in the table against the properties of adaptable framework.

In the table, we briefly describe the approach of each reflective model and illustrate the co-ordination, integration and adaptation of meta objects. The reflective models in Table 5.1 attempt to perform reflection with different design technology in an object-oriented approach. However, they fail to allow adaptability to support the evolution of mission critical systems. They suffer from either the cohesion of objects dependency (highly coupling) between system meta objects and role objects (Lunan [67], Lemos et al [70] and Tramontana [71]), or are limited to anticipated requirements (Walker et al. [68]), making them unable to perform an adaptation for system evolution.

Software evolution and adaptation offer stimulating challenges. The recent research work in reflective models cannot successfully change the objects' behavior in a runtime environment and they fail to address *adaptability* in system evolution.

Related Works in Reflective models	Lunan [67]	Walker et al. [68]	Lead++ [69] / Lemos et al [70]	Tramontana [71]
Description	System meta object monitors a large set of meta objects. Use the dynamic adding and removing meta objects to and from the list for adaptation.	Contextual reflection consists of a meta level. Interfaces realization for reflection. Use parameters in meta object to perform adaptation.	Components are activated depending on computation. Base on objects interactions and associations for adaptation.	Roles are adapted to change the behavior in reflection Use roles management to construct dynamic roles for adaptation.
Meta Object Coordination (meta space)	System meta object at meta level	Contextual reflection as an *adaptor* intercepts method invocations and performs a mapping between interfaces at meta level.	*Strategy object* performs the computation to cooperate the interaction.	*Cooperation manager* coordinates the request for accessing a group of objects and roles participated.
Meta Object Integration	Use *pre list* and *post list* of meta objects	Contextual reflection at meta level adjusts the parameters to pass to a meta object	*Strategy object* as cooperation manager	nil
Meta Object Adaptation strategy	Add or remove meta object to or from the *lists*	Add or remove meta object to or from the *mapping*. *Adaptor* performs operation depending on the previous history of method calls to the object.	Information in *Strategy object* can be changed at run time. Objects interaction depends on the computation with *strategy object*.	*Role manager* expresses the adaptation strategy.
Communication linkage between Meta Level and Base Level	One system meta attaches to a base object at base level – simple attachment	*Adaptor* maps meta interfaces.	Nil	Intercepts method invocation at meta level.
Evaluation on properties of adaptable framework	Cohesive dependency with system meta objects. Can't make changing to the established meta objects at run time. It lacks of adaptability.	Static mapping. Dynamic adaptation is limited to the anticipated requirement. Can't change the meta interface mapping (*adaptor*) at run time. It lacks of extensibility and adaptability.	Can't perform replacement of the meta objects at run time. Computation in strategy object is predefined. It lacks of adaptability.	Objects and roles are closely coupled at run time. Roles can't evolve dynamically at run time after they have associated with the meta objects. It lacks of adaptability.

Table 5.1 Related Works for Reflective Models with Meta Architecture

Chapter 6 Practical Works on Adaptability in Meta Architecture

6.1 Adaptable Frameworks for Systems Evolution

In this chapter, we review the recent work in building an adaptable framework. In the earlier stage of the research for developing frameworks, many researchers focused on building application specific frameworks. Fayad et al. [20] presented a comprehensive discussion of OO frameworks in which they classified application frameworks. Johnson et al [22] described a framework as a reusable semi-complete application that can be specialized to produce custom applications. Schmid [25] classified frameworks as application- or domain-specific. Yacoub et al. [6][59] using design patterns as building blocks, described pattern-oriented frameworks as containing two distinct levels mentioned in Chapter 4: the pattern level and the class level. His approach addressed reuse at the design level and considered patterns thought as micro-architecture elements and not components viewed at the macro level. However, in practice, it was very difficult to maintain the patterns after they have been realized and regrouped at the class level [59].

Buschmann et al. [2] described the Reflection architectural pattern as providing a mechanism for supporting class modification and affecting subsequent functionality in the application. He presented the reading and writing objects in a file but had not described how the adaptation could be carried out in the runtime environment. Grand [14] collected a set of enterprise design patterns but had no further description on relating these patterns as components in an adaptable framework architecture.

More recent research work has focused on providing component-based frameworks [108]. The primary reasons for component production and

deployment are: separability of components from their context, independent component development, testing, configuration and later reuse, upgrade and replacement in running systems. Assmann [72] presented component models of invasive composition. The techniques in invasive composition help components to adapt to reuse requirements. Assmann et al [73] went further and initiated the development of automated component-based software engineering which has since emerged as a field of study in software engineering. Automated component-based software engineering aimed at studying the adaptation of component-based frameworks. Assmann's study identified adaptability for components to allow system evolution at runtime as a critical issue which should be resolved before a component-based approach can make a significant impact on mission-critical software automation.

The unrelenting pace of change that confronts software developers compels them to make their component-based frameworks more adaptable. The following sections show the current trends in framework research which investigate the benefits that the use of the reflective and aspect-oriented mechanisms for managing could gradually bring to system evolution.

6.2 Reflective Languages - Iguana

Iguana [74] is a programming language which is an extension of C++ or Java with Meta-Object Protocol (MOP) features. A Meta-Object Protocol is an interpreter of the semantics of a program that is open and extensible. A MOP determines what a program means and what its behavior is, but it is extensible in that a programmer can alter the behavior by changing pieces of the MOP. The MOP exposes the internal structures of the program to the programmer. MOPs are implemented as object-oriented programs where all objects are meta objects. The Meta-Object Protocol (MOP) of a programming language is an exemplary model for providing fully functional reflective support in language.

In general, MOP can happen at runtime or compile time. The meta objects of runtime MOPs exist while the program itself is executed. The meta objects of compile-time MOPs, however, exist only when the program is compiled. A compile time MOP is a pre-compiler and acts as an interpreter for the program and interprets the program via modifications of the meta-level information in the program before the program compilation. They alter or extend the compilation process.

Iguana aims at generating runtime reflective software instead of being a reflective compiler itself. Iguana [74] uses reflection as a mechanism for implementing dynamically adaptable system components and MOP in Iguana is used to specify the implementation of a reflective object-model as a consequence of reification.

Iguana has been designed to explicitly facilitate dynamically adaptable objects, i.e., objects whose behavior can be adapted at runtime [75].

In an interpreted language of MOP, an interpreter must construct a lot of behavioral information about a program to be interpreted. This information is not used directly by the program; instead it serves as the behavioral information needed by the interpreter to execute the program. In reflective programming, however, reflection occurs in the reflective computation that involves how the program is interpreted. The reflective program can adapt its own interpretation via modifications of the meta-level information.

Iguana adds reflection to a compiled language such as C++ and Java. As an interpreter has already constructed a significant amount of meta-level information, extending the interpreter to be reflective only involves adding support for exposing the meta level information to the base-level program,

allowing the program to influence the decision making process of the interpreter and effect its own behavior through modification of the meta-level information.

Iguana has a utility called *Reification Categories* in an attempt to provide a simple configuration tool to minimize execution overhead by offering developers the opportunity to selectively choose which meta objects need to be reified. Iguana/C++ and Iguana/J extend C++ and Java programming languages with MOP features respectively [74] [75].

Iguana is implemented as a pre-processor which reads in the Iguana source, digests the meta level extensions, makes the appropriate meta level modifications, and then outputs modified codes. Iguana/C++ and Iguana/J have different pre-processors. After the preprocessing has completed, Iguana will invoke the compiler (C++ or Java) accordingly to Iguana/C++ or Iguana/J coding.

As a result, Iguana architecture has strong cohesion with the implementing language. Their implementations of dynamic adaptation are dependent on the infrastructure of the language environment, such as Iguana/J which is closely integrated into the JVM [74]. The architecture is bound to a language and its reflective ability, which, in fact, decreases its flexibility. Apparently, the architecture of Iguana does not support the concept of independence of run time platforms and the semantic integrity of components in a framework. The tool, *Reification Categories* in Iguana, offers reification in framework and does not support the system operations, such as *start/stop*, on meta objects for system administration. The framework architecture of Iguana is language dependent and fails to address *Separation of Concerns* and *Portability*, the two indispensable properties of an adaptable framework.

6.3 Adaptable Models

6.3.1 Adaptive Object Models

Adaptive Object Model (AOM) [76][77] is an architecture intended to provide a meta architecture that allows requirement changes to be performed and immediately reflected at runtime. AOM architectures are usually made up of several smaller design patterns, such as Composite, Interpreter, Builder, and Strategy [1], along with other dynamic patterns such as Type Object, Property, and Rule Objects [78]. By organizing an application using these patterns, we can represent application features, attributes and rules as metadata that can be interpreted in a running system. After this organization, a requirement change can be performed, for example, by replacing the interpreted metadata, which can be stored on the database or in an XML file.

The maintainability problems with AOM's code happen because the adaptive behavior is often mixed with the business logic and GUI code of the application. AOM systems' code is usually very difficult to understand and maintain. The internal structures of AOM are difficult to extend and maintain as agreed by AOM architecture developers [77]. In this case, AOM architecture is not adaptable because it is not easy to include unanticipated adaptive requirements on them and it is not even able to reflect immediately the change requirements at run time.

Aspect-Oriented Programming (AOP) described in Section 5.4.1 has been proposed to make AOM systems simpler to evolve, regarding the inclusion of new adaptive requirements [76]. AOP is used to identify the *cutting points* in the execution flow of a program. The *cutting points* are the places in a program where behavior changes happen. For implementing these adaptive *cutting points*, the concept of dynamic properties dictionary is introduced in Adaptability Aspects Pattern [77] to define and store those adaptive *cutting points*. The dynamic

properties dictionary is also responsible for verifying whether an adaptation should be performed, followed by performing the necessary changes consistently. The use of the Adaptability Aspects Pattern is proposed to avoid code tangling and scattering while implementing adaptability, which can isolate the adaptability actions from the application business logic and GUI code. However, business cases are lacking to illustrate how the concepts of these adaptive aspects can be constructed and applied to the AOM architecture to achieve *adaptability*.

6.3.2 *Dynamic Hyperslices Model*

The Dynamic Hyperslices Model [79] is intended to support the dynamic evolution of non-stop systems which can not be easily taken offline for maintenance due to high costs of their down time (telephone and banking), or environment safety (nuclear plants) or loss of human life (life support systems), etc. The Dynamic Hyperslices Model is proposed. It uses the Hyperspaces approach [80] to decompose the software system into single aspect modules (Hyperslices), and then apply the reflection (meta architecture) along with architectural connectors, to achieve dynamic unit composition and runtime manipulation.

The Dynamic Hyperslices Model is currently under development. The architectural designers have the initial ideas for its implementation. The link between the base and the meta levels is established at load time via `AspectJ` which introduces and initializes the corresponding reference variables at the base and meta levels. `AspectJ` is an extension to the Java programming language.

1 A composed slice is represented by a proxy class at the base level and a composite meta class at the meta level. Instantiation of a composed slice results in instantiations of its components;
2 All calls to the base-level objects are passed to their meta objects. The meta

objects resolve each call in accordance with the composition strategy used and filter it down the composition chain to the resolved primary slice which executes the call.

After the composite classes are instantiated at the meta level, the links between composite objects and proxies are established. Therefore, the composite objects (meta objects) in the model cannot be easily replaced with new objects (evolving version) at run time. Currently, the proposed Dynamic Hyperslices Model only shows the layout of a meta architecture on top of a Hyperspace Model with Hyperslices as components at the meta level and is still not able to support dynamic adaptation for mission critical systems for evolution [79]. The problem for handling new definition of a hyperslice in hyperspace has not been resolved in connection with the evolving schema [79]. Besides, some very important issues for system evolution, such as dynamic integration and management of Hyperslices, have not been addressed in the model. The architecture of our adaptable framework in this thesis aims to resolve the issues for the dynamic integration and management of meta objects for system evolution.

6.4 RAMSES - Reflective Middleware for Software Evolution

Reflective and Adaptive Middleware for Software Evolution of Systems (RAMSES) [81] is a reflective middleware whose aim consists of consistently evolving software systems against runtime changes. This middleware performs two phases in carrying out adaptation.

In the first phase, the RAMSES's meta-level extracts the design information as XMI schemas from the base application and it reifies them at the meta level to constitute the meta-data. In the second phase, the RAMSES's meta-level plans the dynamic adaptation of the base-level system, gets the runtime events, evolves the meta-data against the detected event, checks the consistency, and finally reflects

the modified data to the base level [81].

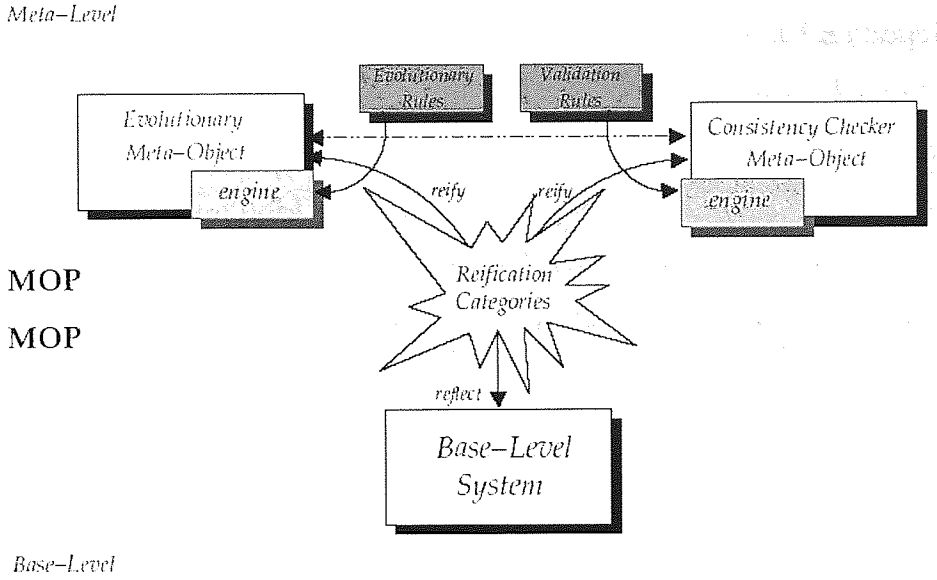

Figure 6.1 RAMSES Architecture [81]

This infrastructure is considered to be dynamically adaptive because changes in the execution environment causes objects, attributes and collaborations to be constructed and modified at runtime to achieve new behaviors not previously defined and instantiated. Figure 6.1 shows the two-layers reflective architecture. The base level is the system that renders self-adapting whereas the meta level is a second software system which reifies the base-level design information and plans its evolution when particular events occur.

RAMSES is starting to work on its prototyping and design XMI schemas for base-level attributes from design information [81]. RAMSES is in the development stage and it has not overcome the problems which arise from the functional domain to infrastructure, such as coordination and integration during evolution, time constraints for meta objects instantiation at the meta level, and configurative administration at the meta level at run time to support adaptability issue.

6.5 Hot Deployment and Hot Evolution in Application Server – J2EE

Most of application servers including both commercial (Websphere, Weblogic) and open-source (JBoss, Tomcat) provide the hot deployment functionality [82], which enables software components (EJB-JAR, EAR, or WAR package) [36] to be plugged and unplugged without restarting application servers. This dramatically improves the productivity of the software development. To be separately deployed and undeployed at runtime, each component is loaded by a distinct class loader. Thus a J2EE application can be dynamically customized per component at run time. Sato et al [82] define J2EE application server as providing "Hot Deployment" .

However, if a new version of a component is deployed for software evolution, instances of the new version cannot be exchanged for *concerned instances* (such as caches, cookies, session objects) of the old version of that component because of the version barrier of current application servers. Such *concerned instances* in EJB objects should be passed to the new component through the shared container of the application server. This is remarkably inconvenient in practice. That is to say, J2EE application servers provide hot deployment but not "hot evolution" . The version barrier makes it difficult to include a copy of a shared class in every component. JBoss [83] provides the UCL (Unified Class Loader) architecture but this architecture prevents different components from including different classes with the same name. As the result, J2EE and JBoss both lack required *portability* as defined in Chapter 2 in their framework architectures for system evolution.

Middleware technologies such as Sun Microsystem's J2EE are slowly moving into the domain of automated component-based software engineering. Their emphasis is typically on generation, such as glue code generation or user-interface generation; support for automated testing, adaptability and quality control is only nascent.

6.6 Reflective Model Driven Architecture - OpenCOM

OpenORB [84] is a pilot implementation of a reflective meta-model OpenCOM. OpenORB is a reflective dynamically reconfigurable ORB middleware platform built on top of OpenCOM. The Object Request Broker (ORB) acts as a central object bus over which each CORBA object interacts transparently with other CORBA objects located either locally or remotely.

CORBA is the middleware that relies on a protocol called the Internet Inter-ORB Protocol (IIOP) for invoking remote objects. Everything in the CORBA architecture depends on an Object Request Broker (ORB). Currently, the lightweight component technology is applied only at the application level on top of middleware infrastructures, which hide distribution and other non-functional concerns from component developers.

In order to address the need for adaptability, the middleware, OpenORB, is built according to a component-based architecture. Furthermore, it is proposed that the middleware is reflective in helping to facilitate and manage run-time changes in component configurations. In other words, it should incorporate structures representing framework aspects of itself and offer meta-interfaces for inspecting and adapting the reified aspects (meta objects). The architecture shows that both middleware and applications should be built from components following the lightweight component model (Object Request Broker).

At this stage, OpenORB supports dynamic composition for distributed applications, but does not provide adaptability of meta objects for system evolution at run time. Recently, OpenCOM has proposed further development for the design of reflective middleware architecture [84, 109]. As challenging new requirements [109] emerge when working with a distributed architecture, these

include decisions in software design, component development, integration, deployment and adaptation at run time. To address these issues, OpenCOM is currently investigating the use of Model Driven Software Development (MDSD) [84]. OMG Model Driven Architecture (MDA) is about using modeling languages as programming language. Programming with modeling languages can improve software productivity, quality and longevity [102]. OMG Model Driven Architecture (MDA) will be adapted in OpenCOM, and currently the main focus of applying MDA is on the design of distributed applications and ways to map them to middleware technology (OpenORB) [84].

A key issue of OpenCOM is to investigate ways to maintain the UML models at runtime and to keep them causally connected with the underlying running system in order to support reconfiguration and reflection [110]. Applying MDA for OpenCOM reflective component framework is now at a primitive stage. More work has to be done to completely identify the variability among configurations of middleware families to support an efficient generation of configurations in the Model Driven Architecture (MDA) [84].

6.7 Summary of the Reflective Frameworks

Recently, framework researchers have identified that adaptability of components in systems evolution at runtime as a critical issue. It should be resolved before any component-based approach can make a significant impact on mission-critical software automation. The software adapation combines aspects in *adaptability* for system development with *Separation of Concerns* and *Portability* to aid the evolution of running systems.

Separation of Concerns provides the innovative composition and consequent maintenance of components in the framework architecture. *Portability* adds flexibility to the framework architecture. This allows it to be independent of programming language and its platform, enhancing the transfer of concerned instances between versions of distributed services in computing.

We have briefly reviewed the recent reflective frameworks and summarize them in Table 6.1. In this section, we categorize the related works in reflective frameworks according to same assessment attributes used in previous chapter: meta objects co-ordination, meta objects integration, meta objects adaptation, and communication linkage between base level and meta level. We also evaluate each reflective framework included in the table against the properties of adaptable framework.

The drawbacks we identified in the recent research framework architectures include: *Customization JVM for Java* in order to meet dynamic adaptation with Java runtime environment (Iguana [74]) and *Complication in extending and maintaining framework internal structure* for new user requirements (AOM [76]).

Customization JVM for Java makes the architecture impractical for implementing adaptable framework with independence of programming language.

Complication in extending and maintaining framework internal structure causes the architecture to be difficult of managing the unanticipated requirements in system evolution. Furthermore, the barrier for the transfer of the concerned instances (caches, sessions) between versions of the framework component (J2EE [82]) also causes failure in portability for the distributed services in software evolution.

Both academic and industrial researchers are aware of the importance of having component adaptation for distributed object-oriented enterprise framework. Reflective frameworks, Dynamic Hyperslices Model (IBM) [79, 80] and RAMSE [81] in Table 6.1, are currently under development to meet the demand of adaptability for software evolution in component-based applications.

Currently, no adaptable framework takes successfully into account the changing of components' behavior for the dynamic reflection at run time. Our thesis aims at providing a reflective framework which can resolve the runtime critical issue of adaptability of components in systems evolution in a distributed computing environment.

Related works in Reflective Frameworks	Iguana [74] Reflective Language	Adaptive Object Models (AOM) [76]	Dynamic Hyperslices Model [79,80]	RAMSES [81]
Brief Description	MOP Programming language extends C++ or Java Pre-processor reads in Iguana source and digests them to the meta level extensions.	Generic design patterns and dynamic patterns are used in building reflective model. *Dynamic properties dictionary* in `Adaptability aspects pattern` stores the adaptive cutting points	Decomposition of software system into simple aspects (*Hyperslices*) Software system can be decomposed into single aspect modules.	Reflective middleware consists of evolving software systems against runtime changes. Two layers reflective architecture: base level is the system renders self-adapting and meta level plans its evolution.
Meta Object Coordination (meta space)	Interpreter in MOP has a significant amount of meta level information.	The grouping of objects in patterns provides immediately reflection at run time.	*Hyperspace* is top level of *Hyperslics*. Composed slice coordinates and consists of a collection of single *Hyperslics*.	(First Phase) Meta level extracts design information and constitutes the *metadata*. (Second Phase) *Metadata* against the detected event to reflect the modified data to the base level.
Meta Object Integration	nil	Objects integration specified in cutting points in execution flow of program.	Dynamic unit composition with *Composition Strategy*	*nil*
Meta Object Adaptation Strategy	Dynamic adaptation by customized JVM	Adaptation defined in *cutting points* of the program flow.	Dynamic adaptation under development	Meta level plans the adaptation and is under development
Communication linkage between Meta Level and Base Level	Iguana exposes the meta level information to the base-level program	*Cutting points* where behavior changes happen.	Architectural connectors - the link established at load time via *AspectJ*	*nil*
Configuration Management	Reification categories to allow developers to choose meta objects to be reified.	Features, attributes and rules are represented as *metadata*.	nil	nil
Evaluation on properties of adaptable framework	Custom *JVM* for Java. Reflection is added to the compiled language and has dependency on the infrastructure of the language [74]. Lack of *portability*.	Limited to anticipated runtime requirements - Internal structures are difficult to extend and maintain [76]. Lack of *extensibility* and *portability*.	Architecture is not capable to handle to support the evolving schema in current stage [80]. Framework is under development. Lack of *adaptability*.	Architecture is work on prototyping and design XMI schemas for base-level attributes for runtime adaptability [81]. Lack of *adaptability*.

Table 6.1 Reflective Frameworks for Software Evolution

PART III

ADAPTABLE ARCHITECTURE OF MELC

Chapter 7 An Adaptable Meta-Based Framework Architecture

In this chapter, we propose the conceptual and physical design of our Meta-Level Component-Based Framework (MELC) architecture which uses distributed computing design patterns as components to develop an adaptable pattern-oriented framework for distributed computing applications. We describe our novel approach of combining a meta architecture with a pattern-oriented framework, resulting in an adaptable framework which provides a mechanism to facilitate system evolution [85, 86]. It is particularly important that our framework possess the five major properties mentioned in Chapter 2: *separation of concerns, adaptability, transparency, extensibility,* and *portability*. They are the criteria for assessing the attributes of our framework. The concept of using a meta architecture to produce an adaptable pattern-oriented framework for distributed computing is new and has not so far been explored elsewhere [86].

This chapter presents the key concept and the internal kernel design of MELC architecture. We will address the several major features in the implementation of the architecture. How does MELC coordinate the meta objects to reduce their interconnection at the meta level (regarding management in meta architecture)? How do base objects get to know about which meta objects they are connected with (regarding reification in meta architecture)? How do meta objects get to know which base objects they are connected with (regarding reflection in meta architecture)? How does MELC handle operations (start/stop) between two levels (regarding separation of controls between two levels)? Can the design of MELC framework be feasibly implemented in an object-oriented programming language such as JAVA (regarding the programming model)? Such questions will be dealt with thoroughly in order in the following sections.

Issues that will be addressed: the identification of distributed computing patterns

for meta components in Chapter 8, the development cycle of an application with MELC in Chapter 9, the installation of meta components and the integration of meta components in Chapter 10, and the adaptability of components in MELC in Chapter 11. They are the technical and core parts in MELC, which will be described step-by-step in full detail.

7.1 Conceptual Architecture of Adaptable Meta-Based Framework

Conceptually, we apply meta architecture to construct an adaptable layer to the pattern-oriented framework in order to make the framework adaptable. In revisiting the three stages in the construction of the framework described in Chapter 2, they are framework construction, framework instantiation and framework application, and we find that the pattern-oriented framework approach can be aligned with them consecutively, as in Figure 7.1.

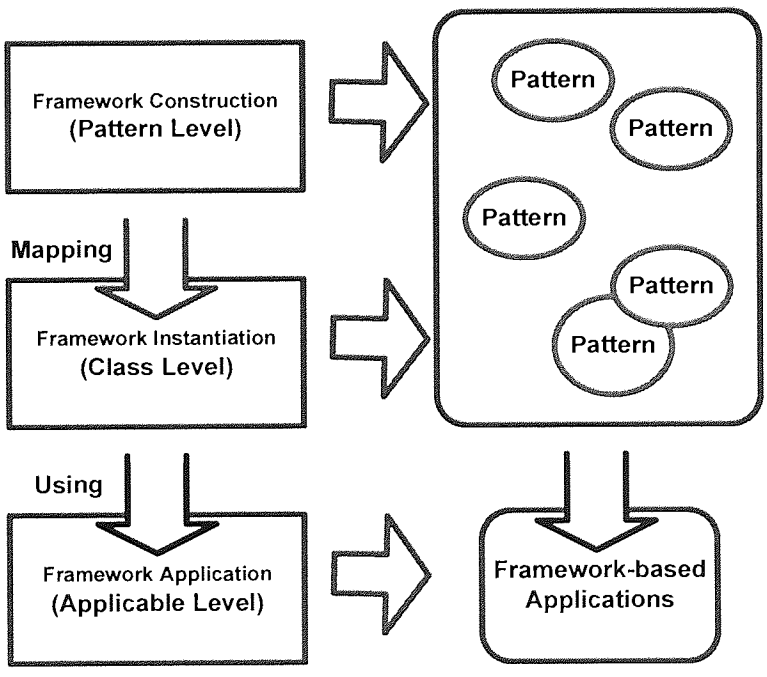

Figure 7.1 Pattern-oriented Framework provides Pattern Level (Framework Design Level) and Class Level (Framework Class Level)

The figure shows that a pattern-oriented framework uses a pattern level which serves as a design layer above the class level. On the left side of Figure 7.1, the rectangles show the levels, and the arrows on the left show the mapping between the framework design level (Framework Construction) to the framework class level (Framework Instantiation). The rounded rectangle on the right side shows the components that have been instantiated by the design patterns, and the arrow on the right shows the usage of components in the framework-based application. For the software quality perspective, the instantiation of well-established design patterns as components in the framework can contribute to the design quality of the constructed framework [59].

Pattern-oriented frameworks have some drawbacks related to *adaptability* as described in Chapter 5. We introduce an adaptable layer into pattern-oriented frameworks by using a meta architecture that creates a higher level of abstraction. The meta level in our framework has (system) components which are based on the well-defined and proven domain specific design patterns. The meta level is designed with a pattern-oriented approach. The components at the meta level are the instantiation of distributed computing enterprise patterns. The base level of meta architecture in our framework contains the application (business) components. The framework architecture separates system functional concerns from business application concerns. Such architecture design adds *Separation of Concerns*, *Extensibility* and *Portability* to our framework.

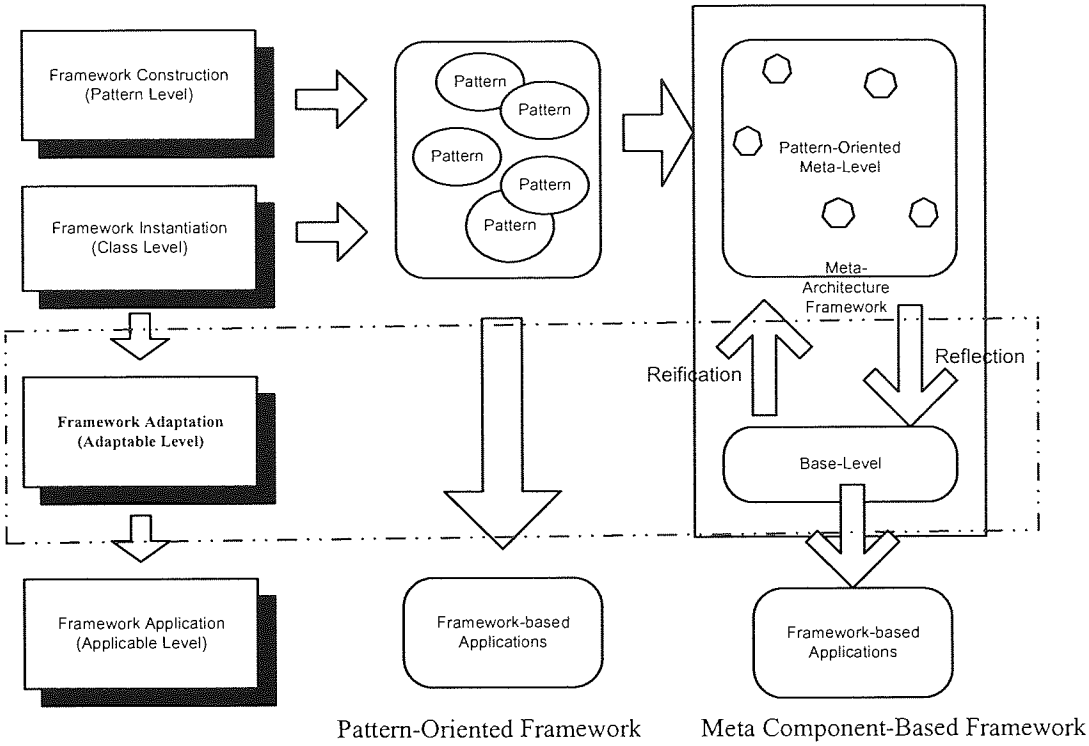

Figure 7.2 Adaptable level introduced in MELC Framework

It is particularly important that, by adding an adaptable layer to the pattern-oriented framework, we will be able to provide a means for *adaptability*, as shown in Figure 7.2. The dotted rectangle in the figure illustrates our proposed new *Framework Adaptation Layer*, superimposed upon the existing framework architecture. The adaptability in MELC is provided by the meta architecture described in Section 5.1. The meta architecture based framework shown on the right side of the figure, which has two arrows: one points from base level to meta level to illustrate the function of reification in meta architecture, which allows meta objects to be accessible by base objects; and the other from meta level to base level to show reflection which illustrates that a change in the meta object can result in changes of behavior in base objects.

MELC supports dynamic adaptation of feasible design decisions in the framework design space by specifying and coordinating meta objects that represent various functional components within the distributed environment.

Our approach resolves the problem of how to allow for dynamic adaptation to enable system evolution which is encountered in most distributed applications. The adaptability in the framework architecture will be further illustrated in next sections and succeeding chapters. The work to develop a meta-based pattern-oriented framework for distributed computing applications in this thesis has emerged as a promising way to meet current and future challenges in a distributed environment.

7.2 Internal Kernel Design of Adaptable Meta-Based Framework

An application can be split into two parts: system behavior and business application. The meta level provides information about system properties and behaviors, and the base level consists of business application logic. The implementation of system functionality is built at the meta level. Any changes in the state at the meta level affect subsequent base level behavior. Figure 7.3 shows the framework architecture where the meta level contains distributed computing patterns and the base level contains base objects (application servers).

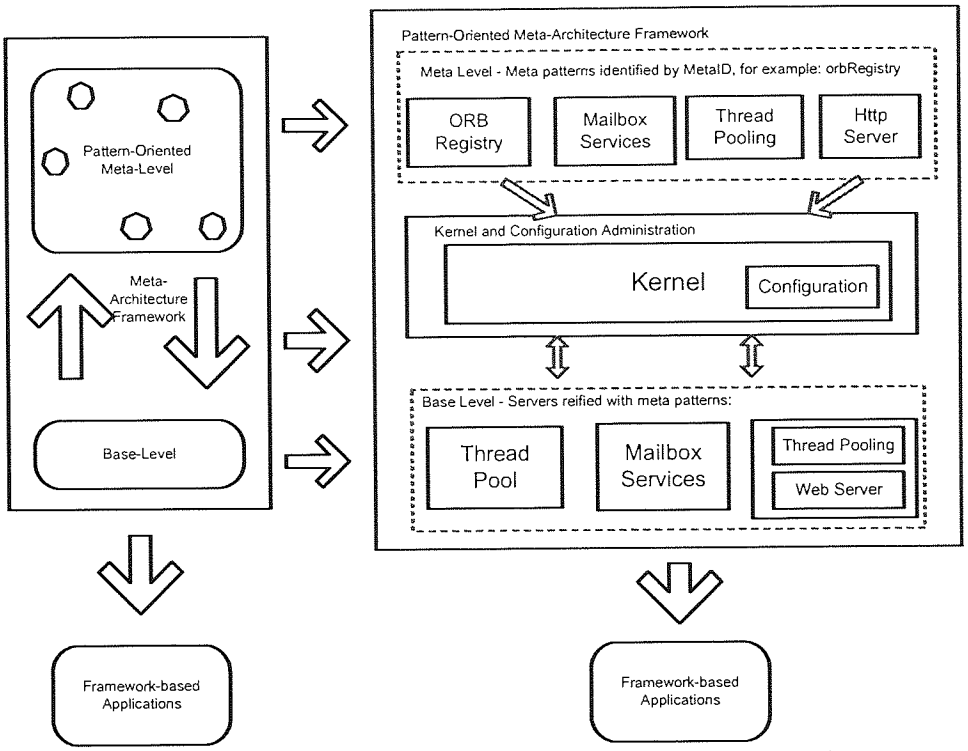

Figure 7.3 Meta Level Component-Based Framework Architecture

Figure 7.3 shows our framework flanked by conceptual architecture on the left and physical architecture on the right. The rectangle on the left of the figure represents *conceptual architecture* which contains two levels: the Pattern-Oriented Meta Level and the Base Level. The top-down arrow and bottom-up arrow shows the reflection and reification respectively happened in the meta architecture. The rectangle on the right of the figure represents the *physical architecture* which has a Kernel to coordinate the interaction of meta level and base level. The figure shows the meta level contains distributed computing components (ORB Registry, MailBox Services, Thread Pooling and Http Server), and the base level has the business servers that have reified the components at the meta level.

The Kernel of the Meta Level Component-Based Framework (MELC), shown in more detail in Figure 7.4, provides the core functions for the adaptation between meta level and base level. It includes: *Meta objects and Meta Space Management*

which handles meta level configuration, *Reflection Management* which provides dynamic reflection from the meta level to the base level, and *Reification Management* which provides dynamic reification from the base level to the meta level.

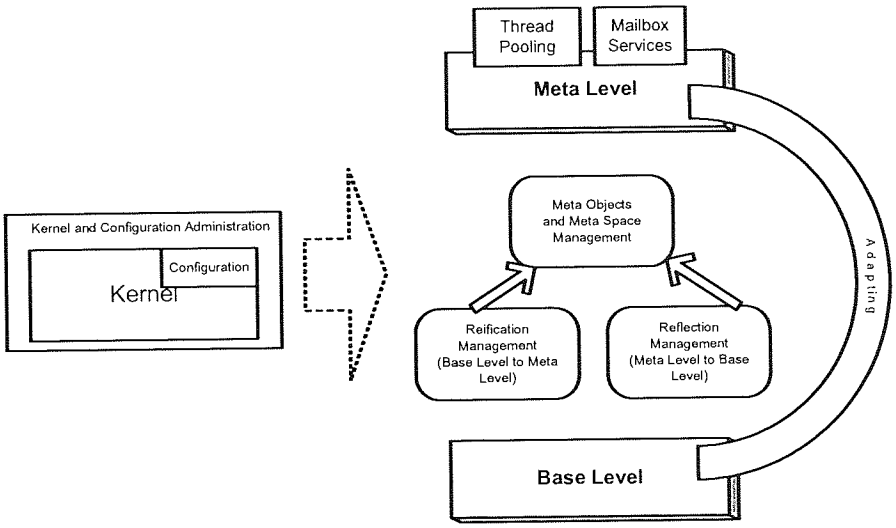

Figure 7.4. Kernel Design in MELC Framework

In our distributed computing framework environment, meta objects are the pattern components, which include ORB Registry object, Mailbox Services object, Thread Pooling object, Http Server object, etc. The meta space in MELC references every meta object and coordinates the interaction between the meta objects to meet user's requests.

Meta objects can be created and added to the meta space after the kernel has been started by the MELC administrator using the management tool which will be further illustrated in Chapter 9. The administrator uses the Kernel Configuration Manager to save the framework configuration, which records the set of meta objects currently running in the system, into a configuration file. Every time MELC is re-started or restructured, it can be initialized by choosing one of the saved configuration files.

Base objects do not have to be attached (reified) to the meta level at the time when the server is started up. They can be attached later by the administrator (on demand). Once a base object is attached to the meta level, it becomes aware of the meta level. For example, if a particular base object is created and reified with a corresponding meta object that provides an implementation of a thread pool, that base object will then be able to provide the functionality of the thread pool to the user's requests. Reflection in our framework allows a base object to perform the functions according to the reifying meta objects in the meta space. As another example, a single base object can be reified with meta objects providing, say, a thread pooling component and a ORB component. The base object will dynamically be reflected to have the behavior of a thread pooling ORB server.

7.3 Reflective Kernel Design in MELC

The kernel in MELC provides the core functions for the adaptation between meta level and base level. Figure 7.5 shows the core functions for the reflective kernel in our framework architecture, which are *Meta Objects and Meta Space Management*, *Reification Management* and *Reflection Management*. They are described and illustrated in the following sections.

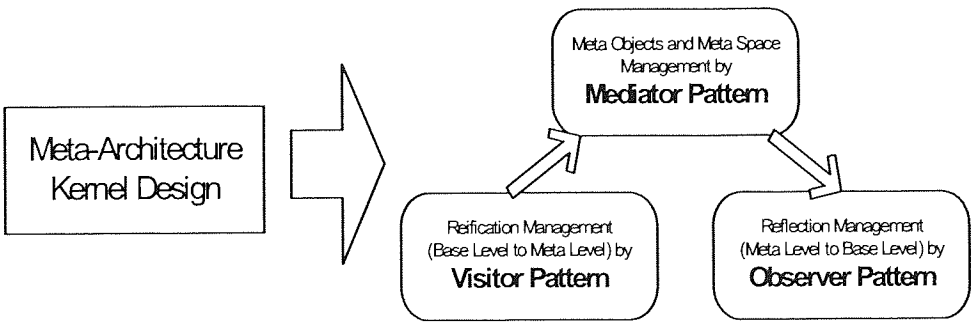

Figure 7.5 Internal Architecture in Meta Architecture

7.3.1 *Meta Objects and Meta Space Management*

This section deals with the question of how MELC coordinates the meta objects to reduce their interconnection in meta space. The Mediator Pattern provides a way of implementing a director which coordinates meta objects. The meta objects are the meta components in MELC. The director in mediator pattern receives the base object requests and redirects them to the appropriate meta objects. We model the kernel function, *Meta Objects and Meta Space Management*, with an implementation of the Mediator Pattern. Figure 7.6 shows the class diagram and the sequence diagram of the function, which are placed on the left and the right respectively in the figure. The class called `MetaSpace` acts as a director which manages a pattern repository to store the instantiation of distributed computing patterns, and coordinates the interaction, reflection and reification of meta components. Meta components are the available meta objects, which we will refer to as its *colleagues*. In addition, the mediator pattern is employed in such a way that the meta components can be added at runtime (plugged in) to `MetaSpace`, or updated (replaced) with the new version of the components in the `MetaSpace` when system evolution happens, or can be removed (unplugged) from the `MetaSpace`.

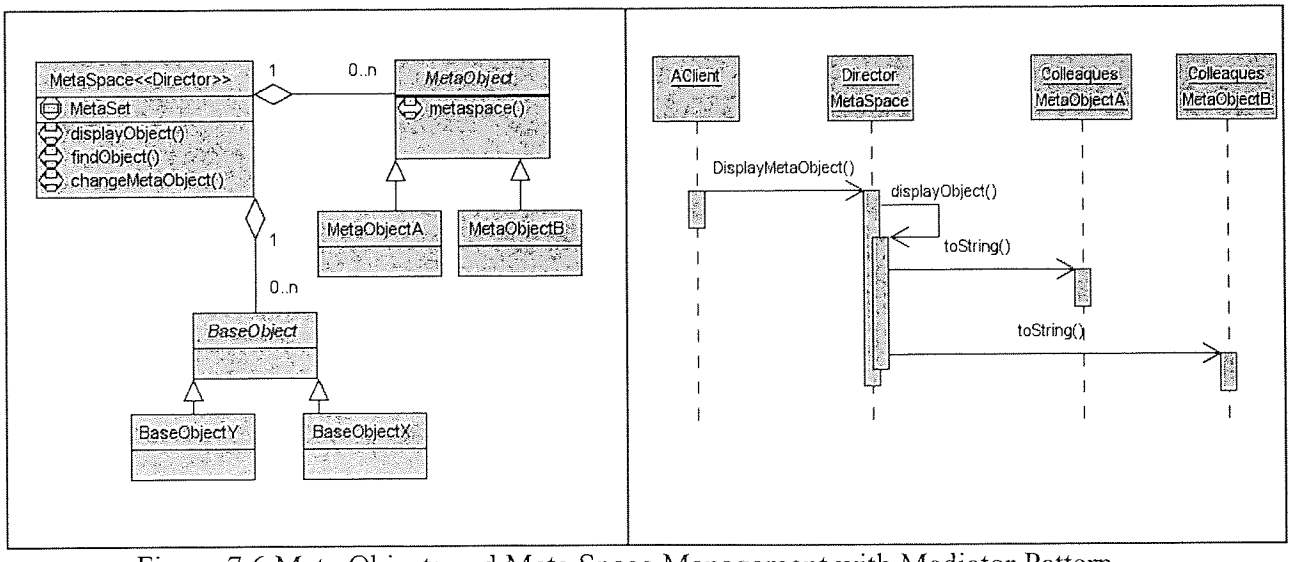

Figure 7.6 Meta Objects and Meta Space Management with Mediator Pattern

The mediator in the MELC kernel architecture is responsible for controlling and coordinating the interactions of meta objects. The `MetaSpace` class functions as a mediator and serves as an intermediary, promoting loose coupling by keeping meta objects in the group from referring to each other explicitly. A mediator replaces many-to-many interactions with one-to-many interactions between the mediator and its colleagues. The one-to-many relationships are easier to maintain and extend. Meta objects only know about the mediator, thereby reducing the number of interconnections at the meta level. Individual objects communicate via the kernel, which serves as their mediator. This prevents them from communicating directly with each other. That is how meta objects interact in the framework. Thus, mediator adds *Separation of Concerns* and *Portability* to our framework architecture.

The sequence diagram on the right of Figure 7.6 illustrates how the objects cooperate to handle a request to display a list of all the meta objects residing in meta space. Such an operation occurs in our administration tool, for example, when the user wishes to select a meta object to reify a base object with. At that point, the tool will present the user with a list of available meta objects to select from. Note how the mediator mediates between the meta objects. Meta objects don't have to know about each other in the kernel; all they know is the mediator, `MetaSpace`. Furthermore, because the behavior of coordination is localized in `MetaSpace`, the interconnections of meta objects are reduced.

The mediator in the framework architecture can decouple colleagues (meta objects), which provides the capability for our framework to update or replace meta components. It makes MELC extensible, adaptable and configurable.

7.3.2 *Reification Management*

This section deals with the question of how a base object establishes its links with meta objects. Reification is the process of making a meta object accessible to a base object. We model the kernel function, *Reification*, with an implementation of the Visitor Pattern. Visitor pattern provides a way of implementing meta object as visitor. Once a base object accepts the visitor (meta object), the base object then may access the reifying meta object. A base object may reify a number of visitors (meta objects). The meta space (the *mediator* described previously in Section 7.3.1) maintains a collection of the reifying meta objects of a base object so that the base object could access them whenever the user request for services of the meta object reaches. For example, a base object, Drawing Pad in Section 5.2, can reify meta objects (ScribbleMeta and ScribbleMeta2) and the Drawing Pad can then listen to events like mousePressed, mouseDragged and mouseReleased for drawing and the choice control to choose a new pen color.

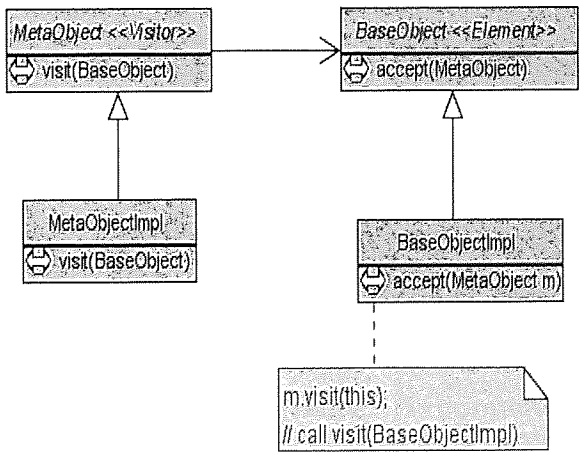

Figure 7.7 Class diagram for Reification Management Modeling with Visitor Pattern

The class diagram of a visitor pattern is shown in Figure 7.7. The visitor pattern in MELC performs *Reification* by providing visitors (meta objects) to be reified by

the `accept` operation of the base object. Once the visitor has been accepted by the base object, the meta space (the class `MetaSpace` shown in the class diagram in Figure 7.6) stores the link between the base object and the meta object. The base object can then access the reifying meta object via meta space whenever its services are requested by a client. For instance, in an E-Bookshop application, the base object *E-bookshop* reifies the meta component `Subscriber` at meta level and *E-bookshop* can perform the business function, *Customer Subscription*, with the meta space in MELC.

Meta space keeps track of the links between base object and its reifying meta objects. Each meta object has an identifier that is used to uniquely identify the meta object in meta space. The process of identification helps to identify the reifying meta objects of the base object. For example, when a user's request for customer subscription reaches the base object *E-bookshop*, the base object searches for the identifier of the meta object, `Subscriber`, in the repository of the base object *E-bookshop* in meta space. If the identifier is found, the base object will issue a request to the meta object with that identifier and *E-bookshop* can perform the functions of the meta object, `Subscriber`, accordingly.

Visitor pattern in Figure 7.7 provides a design structure that allows adding new visitors (meta objects), simply by having a new subclass in visitor class hierarchy – a subclass of `MetaObject`. In other words, we can define a new distributed computing component (meta component) with the kernel architecture which has no effect on existing base objects and meta objects in system. For example, the meta object reified by *E-bookshop*, `Subscriber`, is a subclass of `MetaObject`. We also can add a new meta component, say `ThreadPool`, by extending `MetaObject`. Thus, the kernel of MELC allows *extensibility* at the meta level.

7.3.3 Reflection Management

This section deals with the question of how the changes of a meta object reflects its connected base objects. The meta architecture in MELC provides the adaptability for system evolution. MELC has the ability to replace meta objects when system evolution happens. Reflection in MELC provides a mechanism to account for changes in the description at the meta level which affect the behavior of described objects at the base level. This approach resolves the problem of adaptation in the framework and we will describe how the mechanism is implemented as following.

We employ the Observer Pattern in the MELC kernel to resolve the interactions resulting from the need to implement reflection between the two levels.

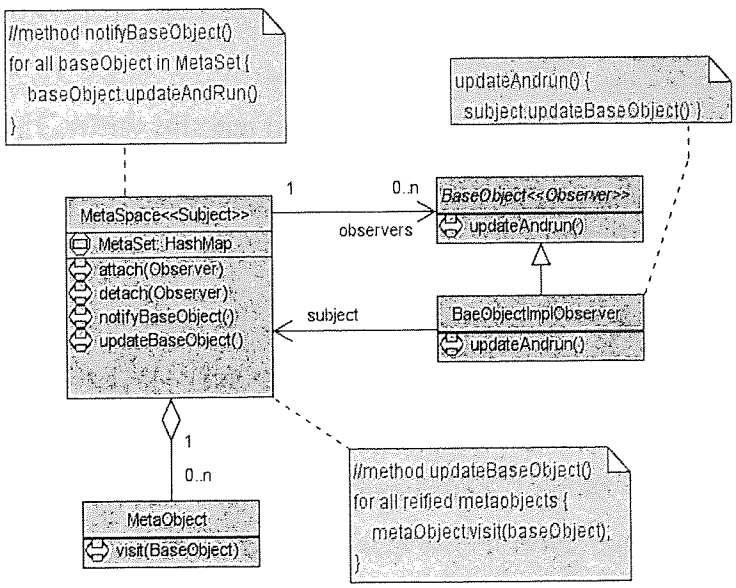

Figure 7.8 Class diagram for Reflection Management Modeling with Observer Pattern

The key classes in the Observer Pattern shown in the Figure 7.8 are *Subject* and *Observer*. Observer Pattern has *Subject* which knows its *Observers* and notifies its *Observers* whenever there is any change in its state. The class diagram in Figure

7.8 illustrates that the observers are the base objects and the subject is the meta space which has a repository (a hash table) to keep track of the reference of each meta object. The subject, `MetaSpace`, is considered as a change manager which maintains and updates the association between the identifier of a meta object with the reference to the meta object itself whenever it receives a request for a change. The changes in the set of meta objects also trigger updates (update actions) of the links between base objects (observers) and their reifying meta objects. Using the observer pattern to implement our reflective framework, all base objects (observers) can access and perform the functions of the meta objects stored in the meta space (subject).

MELC employs the observer pattern to provide *adaptability*, the dynamic behavior changes (the reflection) at run time, for the framework application whenever meta objects are updated (replaced) for system evolution.

7.3.4 Separation of Controls between two levels

This section deals with the question of how MELC addresses the separation of controls between application level (base level) and system level (meta level). MELC provides `start` and `stop` controls at both the base and meta levels and we call them *application level operations* and *system level operations* respectively. The former allows the MELC to start or stop the operation of a reified meta object in a specific base object, whereas the latter allows MELC to start or stop operation of a meta object and affects all those base objects which have reified the meta object. The *application level operations* and *system level operations* perform separately on two levels.

The Proxy Pattern implements *proxy* which serves as a surrogate for access to an object and is applicable whenever there is a need for a sophisticated reference to

the object [1]. Figure 7.9 shows the class diagram of Proxy Pattern in MELC Kernel.

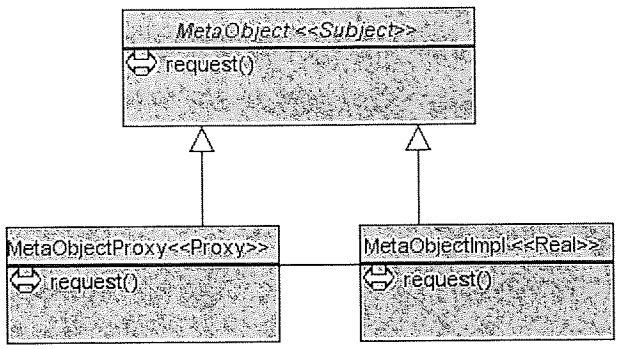

Figure 7.9 Class diagram for Separation of Controls with Proxy Pattern

In Figure 7.9, `MetaObjectImpl` is the real subject and `MetaObjectProxy` is the proxy subject. A user sends a request to a proxy subject which hides the fact that the real subject exists. The proxy object is responsible for encoding a request and its arguments and for sending the encoded request to the real subject. A proxy meta object at the base level controls access to the real meta object at the meta level.

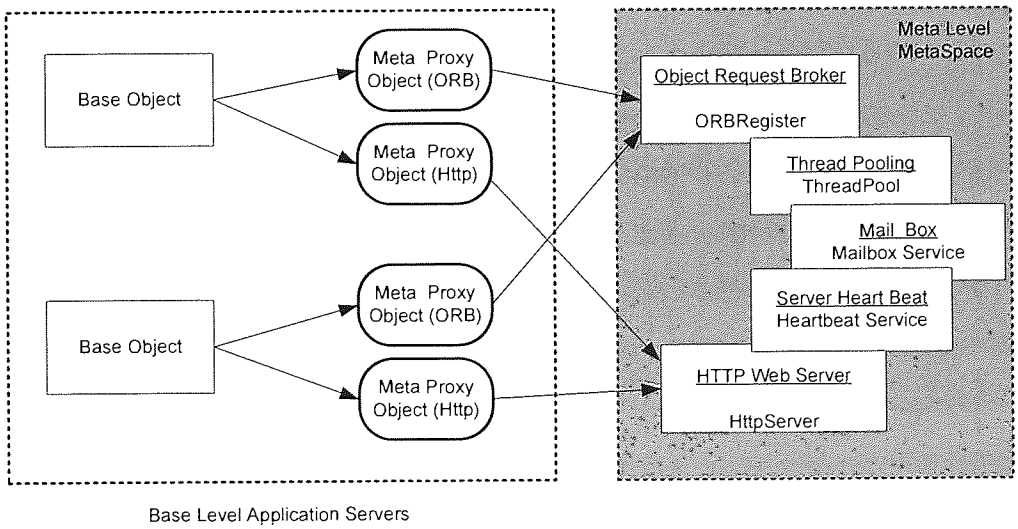

Figure 7.10 A proxy meta object is a local representative of meta object at base level

Figure 7.10 illustrates the use of the proxy pattern in our framework. Using the proxy pattern, we avoid direct references between a meta object and base objects. Each base object talks to a reifying meta object via the unique proxy associated with the latter. The proxy meta object is a local representative at base level of a real meta object. In MELC, associating a base object with meta proxy objects at the base level allows instances of the meta objects to be accessed by the base object. The association is constructed at the time when the base object reified the meta objects. Figure 7.10 shows that a base object can associate with many meta proxy objects and each meta proxy object is a local representative of a meta object.

By using the proxy meta object, the `start/stop` operation of a reified meta object at base level (application level) in the meta space only affects the base object involved and the meta object at the meta level (system level) still provides services to other base objects in the system. Using the proxy pattern in the MELC architecture provides flexibility in adding or removing the association of meta objects and decoupling base objects from the meta objects.

7.4 MELC Programming Model

This section addresses the feasibility of implementing our MELC framework using an object-oriented programming language. In this section we implement the design of our reflective kernel described in previous sections. We have illustrated our kernel modeled with patterns, *Mediator, Visitor, Observer* and *Proxy,* which allows our framework to be derived from the language-independent reflective architecture [85].

The challenges involved in implementing MELC are: a) the decoupling of base objects and meta objects, which is able to add adaptability to the framework in order to allow meta objects to evolve individually; b) no cohesive dependency exists among meta objects and base objects, which allows the meta objects (meta components) to evolve while maintaining compatibility with the base level; c) the kernel has the full operational controls on meta objects and base objects, such that the operational controls can be carried out at system level (meta level) and at application level (base level) separately at run time.

Structural Categories	**Class**
base object interface	`melc.lib.kernel.BaseObject`
base object implementation	`melc.lib.kernel.BaseObjectImpl`
meta object interface	`melc.lib.kernel.MetaObject`
meta object implementation	`melc.lib.kernel.MetaObjectImpl`
meta object protocol	`melc.lib.kernel.MetaObjectProxy`
Meta Space Management	**Class**
meta space	`melc.lib.kernel.MetaSpace`
meta space manager	`melc.lib.kernel.Modules`
Distributed Communication	**Class**
Handoff target object	`melc.lib.kernel.TargetObject`
socket wrapper object	`melc.lib.kernel.BTargetObject`

Table 7.1 MELC Reflection Categories and Meta Object Classes

The programming model for MELC kernel management is concerned with the creation of meta objects and base objects, and the reification and de-reification of meta objects in meta space. Our MELC architecture provides programming interface classes (API) for meta components structure and management. The main API classes are listed in Table 7.1.

7.4.1 *Instantiation of Meta Objects – Meta Level Programming*

The key challenge in MELC is the decoupling issue between the base and meta levels in the framework. A meta component in MELC is developed by implementing the abstract class, `MetaObjectImpl` from the package `melc.lib.kernel`. The instantiation of a meta component is a meta object in meta space.

`MetaObjectImpl` has two methods: `runWork()` and `getMetaProxy()`. The method `runWork()` is invoked by the internal thread in the meta object at the time of the meta object instantiation. The method continues to accept new requests and processes the requests according to the process defined by the meta component. As long as no stop operation of the internal thread has been issued in meta object, the method `runWork()` continues to accept and processes the requests. The method `getMetaProxy()` is invoked when meta object is reified by a base object. It constructs the meta object proxy at the base level as local representative of the meta object. As we have mentioned in Section 7.3.4, meta object is the real subject and its meta proxy object is the proxy subject. The meta proxy object provides a local representative at base level of a meta object. A user sends a request to a proxy subject which hides the fact that the real subject exists. Each base object has the proxy for every reifying meta object to talk to. Meta proxy object at the base level controls access to the real meta object at the meta level.

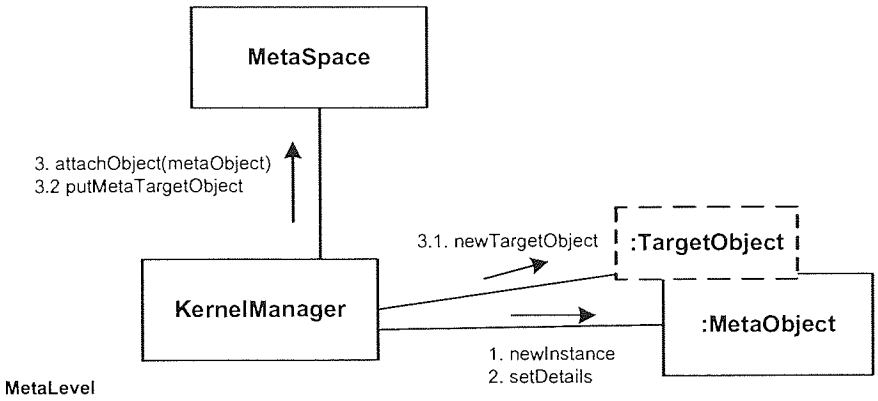

Figure 7.11 Collaboration Diagram for Meta Object created by MELC Kernel Manager

Figure 7.11 illustrates the collaboration between `KernelManager`, `MetaObject` and `MetaSpace`. The following coding is just an example to show how Kernel Manager can easily construct a meta object and the hard coded details of meta object in the coding are input by the user on the user interface of the Kernel Manager.

```
MetaName = "HttpServer.class";
MetaType = "httpserver";
MetaObjectImpl mobj =
    MetaObjectImpl)Class.forName(MetaName).newInstance();
mobj.putClassType(MetaType);
MetaSpace.attachObject(mobj, MetaType)
```

`KernelManager` in Figure 7.11 instantiates the meta objects and attaches them to the meta object repository in meta space. For example, to create a meta object, `HttpServer`, `KernelManager` firstly instantiates the meta object with `newInstance()` by using the given meta object name `MetaName`, and then sets the meta type `MetaType` such as `httpserver` for `HttpServer` and attaches it to meta space.

The `MetaName` and `MetaType` are two character strings and the combination of these two forms the meta object identifier (`MetaID`). The meta object identifier

can uniquely identify the meta objects residing in the meta object repository which is implemented by using a hash table.

New requirements can be implemented by extension of the existing meta object and overriding the existing methods within the original system (we may override the calling methods in `runWork()` of the meta object). The meta components at meta level are open-ended for extension. Thus, our reflective architecture enables the framework to maintain information about itself and use this information to remain extensible.

The target object in Figure 7.11, `TargetObject`, is the connector to pass requests (`Runnable` objects) from base level to meta level. Kernel Manager constructs the target object at the time the meta object attaches to meta space. The `TargetObject` acts as a handoff box for data (object) passing. The requests being sent to a `HttpServer`, for example, are the HTTP Requests from clients. When a client sends a HTTP request to HTTP server, the request will firstly be intercepted by the HTTP server's meta object proxy at the base level. The meta object proxy then transforms the request to a runnable object and places it in the handoff box of the `TargetObject` for the HTTP server. When the meta object, HTTP server, is available for processing, the request in the handoff box will be removed by the thread of the `runWork()` method of HttpServer and the HTTP server will process the request. The meta object proxy and its target object provide a means of separating the base level from the meta level in MELC. Using the proxy pattern in this way to solve the key challenge of decoupling the two levels provides a mechanism for adaptability within the MELC framework.

7.4.2 *Instantiation of Base Objects – Base Level Programming*

Another key challenge in MELC architecture is to provide a simple and uniform way to construct a base object (for example a business server) in the framework. In Java coding, a base object is an instance of the class `BaseObjectImpl` from the package `melc.lib.kernel`, which have functionalities like accepting meta objects while reification, finding the corresponding meta proxy objects, and checking the status of reifying meta objects.

Figure 7.12 illustrates the collaboration between `KernelManager`, `BaseObject` and `MetaSpace`.

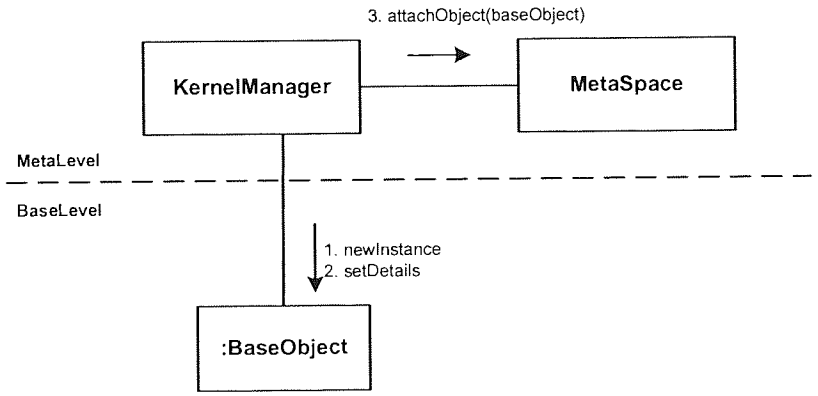

Figure 7.12 Collaboration Diagram for Base Object created by Kernel Manager

The following coding is just an example to show how Kernel Manager can easily construct a base object and the hard coded details of base object in the coding are input by the user on the user interface of the Kernel Manager.

```
BaseName = "E-Bookshop";
BaseObject bobj = (BaseObject)
    Class.forName("kernel.BaseObjectImpl").newInstance();
bobj.putObjectName(BaseName);
MetaSpace.attachObject(bobj);
```

For example, to create a base object E-Bookshop in meta space, `KernelManager` firstly instantiates the base object with `newInstance()`of the class `BaseObjectImpl`. It sets the base object name with `E-Bookshop` which is input by user and is supposed to be unique in the framework, and then informs meta space the existence of the base object in framework by the method, `attachObject()` of meta space.

After the execution of the coding, the object of the business server, *E-Bookshop*, has been constructed and registered in meta space. It meets the requirements of simplicity and ease of construction of the business server at the base level in MELC. You may note that, so far, no meta objects (distributed computing components) have been reified by the base object *E-Bookshop* yet. Nevertheless, they may be required by the business application. The reification of meta objects will be described in next section.

7.4.3 Reification of Meta Objects - Meta Space Programming

In the process of reification of our MELC framework, the base object (`E-Bookshop`) can easily reify the meta objects stored in the repository of meta space. The process of reification of a meta object in MELC is shown in Figure 7.13. The challenge for the implementation of the reification process in our framework is to avoid having any object reference dependencies within the meta space, so as to let meta objects evolve independently. We are doing this because referring to objects directly means we cannot replace them in the runtime. Removing direct references means adding indirect references so a second challenge is to avoid adding too much overhead in the implementation of those indirect references.

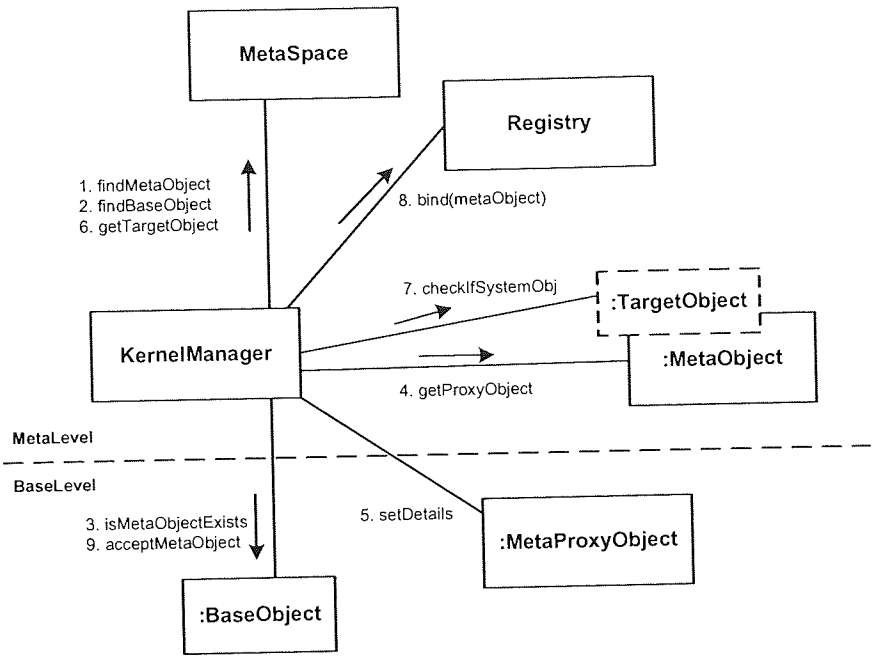

Figure 7.13 Collaboration Diagram for Base Object Reification of a Meta Object

In the process of reification, `KernelManager` in Figure 7.13 firstly verifies the existence of the base object and the required meta object. It also verifies the existence of the required meta object as to whether it has already been reified in the base object. After that, meta object generates its `MetaObjectProxy` as a localized agent (interface protocol) which is used to redirect requests received by an object at the base level to its corresponding meta object at the meta level. The meta object information, `MetaType` and `MetaName`, are used as meta identifier and assigned by the developers. They are stored in the meta proxy object. Meta type is the type of a meta component and meta name is the name of a meta component. `MetaType` and `MetaName` are strictly for identification of a meta object and are not used to as a means of implementing inheritance. For example, `FixedThreadPool` and `FlexibleThreadPool` are two meta names of two different thread pooling components in distributed computing, and they both belong to a meta type called `Pooling`. The base object (`E-Bookshop`) can refer to the meta object (fixed thread pooling) using `Pooling` and `FixedThreadPool`, the `MetaType` and `MetaName` pair, and does not refer to

the object reference of the meta component. At this moment, the proxy object is ready to provide references to both the base object and the meta object.

Instead of using meta object references, the `MetaType` and `MetaName` information kept in the base object are used to refer to a meta object in meta space. The advantage of not using object references for referring meta objects allows the replacement of meta objects in meta space in a run time environment without system reference bounding in some programming language such as C++ and JAVA. The design provides one of the key functionalities, adaptability, to the MELC architecture.

In order to have object distribution previously mentioned in Chapter 3, `KernelManager` binds the meta object to the Object Request Broker `Registry` (ORB) with its meta identifier as binding key for the meta object for remote accesses. The distributed objects in the ORB Register can be accessed seamlessly and transparently by a remote client. The implementation of the Object Request Broker will be described in details in Chapter 10.

In fact, meta objects can be de-reified after they have been reified to a base object. The JAVA code is almost the same as the reification process mentioned except the meta objects are de-registered from the base object.

Table B2 in Appendix B presents the essential methods for `MetaObject`, `BaseObject`, `MetaObjectProxy` and `Registry`, and they are used by the Kernel Manager for the meta reification management.

The program coding for implementation of meta reification management is extracted in Section C.11 in Appendix C for reference.

7.4.4 Application Level Control Operation - Base Level Programming

The MELC framework is designed to address a challenge which is *Separation of Concerns* in the framework between application level and system level. The framework provides `start` and `stop` controls at both the base and meta levels. `Start` and `stop` at the base level are *application level operations* and at the meta level are *system level operations*. Control at the application level allows the `KernelManager` to start or stop the operation of a reified meta object in a specific base object. Control at the system level allows `KernelManager` to start or stop operation of a meta object which then affects all those base objects which have reified that meta object. We will now give an example of application and system level control using in e-bookshop system.

In MELC, each base object owns its own collection of system functions (meta objects) by reifying those meta objects. For example, the base objects, `E-Bookshop` and `E-Banking`, are two separate business servers and they both have reified the meta object `HttpServer` because each server needs to receive HTTP requests. The `HttpServer` of `E-Bookshop` can be started or stopped at the application level without affecting the operation of the `HttpServer` used by `E-Banking`.

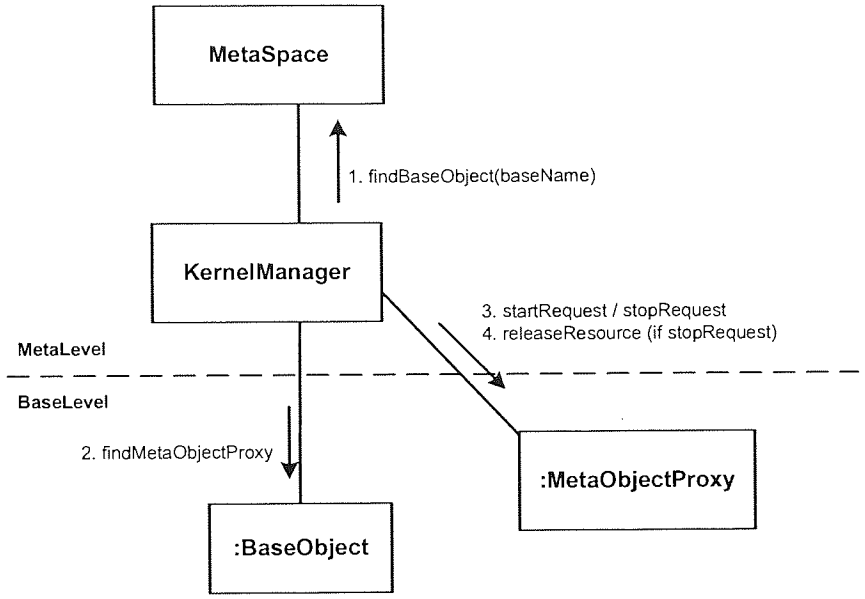

Figure 7.14 Collaboration Diagram for Start/Stop operations at the application level

A collaboration diagram showing start / stop operations at the application level is shown in Figure 7.14. The following coding shows how the `KernelManager` can easily trigger the `start / stop` operation at the application level:

```
// Start a Meta Proxy Object
BaseObject bobj = MetaSpace.findBaseObject(bobjName);
MetaObjectProxy bpobj = bobj.findMetaObjectProxy(mobjType, mobjName);
bpobj.startRequest();

// Stop a Meta Proxy Object
BaseObject bobj = MetaSpace.findBaseObject(bobjName);
MetaObjectProxy bpobj = bobj.findMetaObjectProxy(mobjType, mobjName);
bpobj.stopRequest();
bpobj.releaseResource();
```

The `KernelManager` in Figure 7.14 firstly finds the base object registered in `MetaSpace` using its base name (eg. `E-Bookshop`). Then it retrieves the `MetaObjectProxy` which is the localized representative of the meta object at the base level. `KernelManager` issues the `start` request or the `stop` request to

the proxy object. In addition, for a `stop` request, `KernelManager` will also release the resources occupied by the `MetaObjectProxy` and stop the threads running in the proxy object.

By using the `MetaObjectProxy`, the `start/stop` operation at the application level only affects one base object and the meta object at the meta level continues to provide services to other base objects in the system.

7.4.5 *System Level Control Operation - Meta Level Programming*

In MELC, the `start` or `stop` meta operations at the Meta Level are very different from that at Base Level. Each meta object is a meta component which may be providing services to many different base objects running concurrently in MELC. For example, `HttpServer` provides services to both `E-Bookshop` and `E-Banking`. The `KernelManager` can start or stop a meta object at the system level (meta level), which will affect all base objects that have been reified with it and are therefore using its services. However, the operation will not affect other meta objects such as `MailBox` and `Retransmission` because they are completely separate from HttpServer and would need to be stopped or started in their own right.

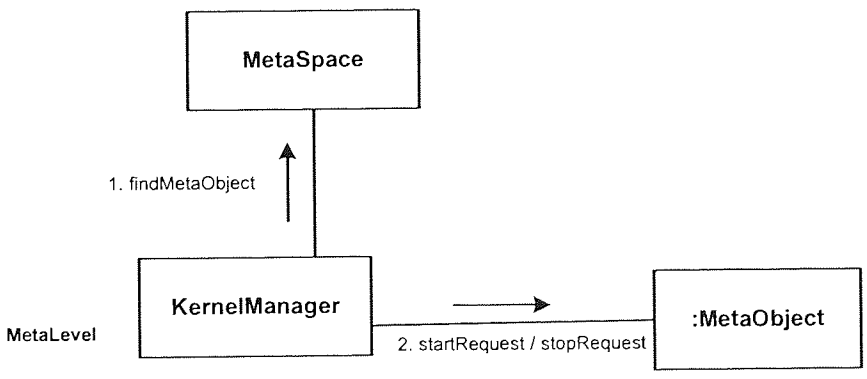

Figure 7.15 Collaboration Diagram for `Start/Stop` operations at the system level

A collaboration diagram showing the `start / stop` operations at the system level is shown in Figure 7.15. The following coding shows how the Kernel Manager can easily trigger the `start / stop` operation at the system level:

```
// Start Mete Object
MetaObject mobj = MetaSpace.findMetaObject(mobjType, mobjName);
mobj.startRequest();

// Stop Mete Object
MetaObject mobj = MetaSpace.findMetaObject(mobjType, mobjName);
mobj.stopRequest();
mobj.releaseResource();
```

The `KernelManager` in Figure 7.15 firstly finds the meta object in `MetaSpace` using its meta type (`MetaType`) and meta name (`MetaName`). Then it issues the `start` or `stop` request to the meta object. In addition, for a `stop` request, `KernelManager` will also release the resources occupied by the meta object and stop the threads currently running in that meta object.

7.5 Summary

Our reflective architecture in MELC enables the framework to maintain information about itself and use this information to remain changeable and extensible. This chapter shows how we model the reflective kernel in our MELC with generic patterns (mediator pattern, observer pattern and visitor pattern). It also describes the implementation of meta objects and base objects with the libraries which we have developed in MELC framework.

Meta space provides the reflection in the architecture. In order to maintain and manage semantic integrity, the MELC framework constructs the components in

the first place and maintains them at the meta level over time as the components in the framework evolve.

The design and implementation of MELC meets the key challenges of a reflective framework described in the Chapter 2. They are *Portability, Extensibility, Separation of Concerns* and *Adaptability*. We summarize them as follows:

1. *Portability in Framework Architecture*: The reflective architecture in MELC is modeled with patterns. We apply the *mediator* pattern for coordinating meta objects in meta space, the *visitor* pattern for reification of meta objects at base level, and the *observer* pattern for reflection of base objects at base level. Thus, the implementation of MELC architecture can be independent of platform and programming language. MELC can be implemented with basic Java classes and have no native code library for low-level support to interact with Java Virtual Machine (JVM) operations nor intercession mechanisms to alter the contents of internal data structure in JVM.

2. *Extensibility at the Meta Level*: MELC Reflective Architecture provides base objects the decoupling with meta objects in meta space. We applied the proxy pattern implicitly to decouple the association of base level and meta level. This indirection reduces the constraints on both, so the levels can be developed independently.

3. *Adaptability for Meta objects*: No cohesive dependency exists among meta objects and base objects. This allows the meta objects (meta components) to evolve while maintaining compatibility with the base level.

4. *Separation of Concerns at levels*: The Kernel of MELC has the full operational controls on meta objects and base objects. The operational controls can be carried out at system level (meta level) and at application level (base level) separately at

run time.

Our kernel design and implementation of MELC meets the challenges of the reflective framework. The installation, integration and reification of distributed computing patterns at meta level will be discussed thoroughly in succeeding chapters.

Chapter 8 MELC – Distributed Computing Patterns

8.1 MELC - Distributed Computing Patterns

The meta components at the meta level in MELC are the distributed computing patterns and they are the domain specific functions in MELC. In this chapter, we try to identify patterns for distributed computing services by looking for typical events or constructs found during the development of distributed computing applications.

In general, design patterns can exist at many levels, from very high-level business solutions to technical system issues. We focus on the distributed computing patterns proposed by Grand [14]. The instantiation of those distributed computing patterns are the meta components in MELC.

We identified 9 common and recurring problems in developing distributed computing applications such as E-Bookshop and E-Banking [88, 89, 90]. The problems occur in a distributed server such as thread pooling, retransmission, heart beat, mail box, object request broker and registry for object distribution.

We identified the patterns by reading the catalogs of patterns [1, 14, 111] – design community and literature, to find appropriate patterns that are applicable to the problems that we have identified. These may not be a complete list of all the problems and their solutions pertaining to the patterns. In time, more distributed computing problems will be encountered in development. The solutions to the problems will be the raw materials for hatching new patterns for our framework.

The patterns we found in this section will be put into the context of MELC. The following are the problems encountered and the recommended pattern solutions:

(1) Global Object Identifier Problem

If an object is common to multiple environments, we need a specific way to uniquely (globally) identify the object in each of these environments. We must be able to distinguish objects even if their attributes have identical values. For example, personal computer workstations in offices may be identical in model, amount of memory, CPU type, and all other attributes. However, it is important that each personal computer workstation has its global identifier in a company.

Most environments that manage objects use the physical location of an object as its unique identifier. A physical location is not in general unique between different environments. Over time, an environment may reuse the same location to store different objects. If an object is common to multiple environments, then we need a common way to uniquely identify the object in all of those environments.

Pattern Solution: Object Identifier Pattern

A solution to the above problem has been proposed [91] and the pattern is called "The Object Identifier Pattern" . This pattern can uniquely identify an object that exists in multiple environments. The method is to assign a globally unique identifier to the object, allowing it to have a unique identity when it is shared between programs or databases. Each shareable object has a globally unique identifier that can be used to unambiguously determine the object's identity.

(2) *Objects for trusted and un-trusted clients Problem*

An object will have trusted and un-trusted clients. An un-trusted client should satisfy a security check before the object answers a request from it. The object should not burden trusted clients with the expense of security checks.

We may have a class that is part of a trusted protection domain that offers access to highly sensitive services. Before an instance of the class satisfies a request from one of its un-trusted clients that will cause it to access one of those sensitive services, the un-trusted client which makes the request must satisfy a security check.

Pattern Solution: Protection Proxy

A solution to the above problem has been proposed [1] and the pattern is called "The Protection Proxy Pattern" . Malicious objects are objects that keep trying to discover and call methods of the environment object, and services they are not supposed to access. In order to forestall malicious objects attempting to violate the integrity of other objects by using reflection or other means to access restricted methods or variables, the pattern requires other objects to access sensitive objects through a proxy that limits access based on security considerations.

(3) *Server/Client Communication Problem*

Programs that communicate with each other through socket-based connections play one of two roles in the establishment of a connection: a client application initiates socket connections with a server, and a server

application waits for clients to initiate connections with it.

We expect the programming code should follow a consistent way to apply server socket in their communications.

Pattern Solution: Server Socket

A solution to the above problem has been proposed [96] and the pattern is called "Server Socket Pattern" . This pattern provides a connection that constructs a server socket which is bound to a given port number, and requests that the operating system queue up connections on the program's behalf.

After a connection is established, programs in both the client and server roles interact with the connection in much the same way. For example, File Server and HTTP Server can be implemented with the pattern. The basic logic that server programs use to manage the establishment of connections is consistent from one server program to the next because they all need to solve the same set of problems.

(4) *Location Changes of Shared Service Problem*

Objects need to find a way to contact other objects which are known only by name or by the service it provides.

Instances of a class exist to provide a service to other objects. From time to time, we may need to change the location of shared-service-providing objects. On the other hand, we want such configuration changes to be transparent to the clients of the service-providing objects.

Pattern Solution: ORB Registry

A solution to the above problem has been proposed [94] and the pattern is called the "ORB Registry Pattern" . This pattern provides a service that takes the name of an object, service, or role and returns a remote proxy that encapsulates the knowledge of knowing how to contact the named object.

Clients share services of the service objects. Service objects register themselves with a registry object. The registration consists of the service object passing its name and a remote proxy object to the registry object's bind method. The remote proxy encapsulates the knowledge of knowing how to contact the named service object. The registration remains in effect until the name of the service object is passed to the registry object's un-bind method. While the registration is in effect, the client can use the object's name to activate the registration object's lookup method, which in turn returns the proxy to the client.

This pattern provides a layer of indirections that determines which service object a client object can use. Client objects are able to access service objects without having any prior knowledge of where the service objects are. That means it is possible to change the location of service objects without having to make any changes to the client classes. However, client and service objects are required to have prior knowledge of where the registry is.

(5) *Server Multi-tasking Problem*

Many servers are often presented with a steady stream of tasks that must be performed in their own thread. Creating a thread is a relatively expensive operation, both in terms of time and memory. We generally want to avoid the expense of creating a thread for each task by reusing threads.

We expect there is an optimal number of threads that a server should be running at one time to efficiently provide supports for multiple tasking and, on the other hand, to cut down the expensive operation of creating new threads.

Pattern Solution: Thread Pool

A solution to the above problem has been proposed [95] and the pattern is called "Thread Pool Pattern" . This pattern provides a service that keeps a pool of idle threads. When a thread finishes a task, it is added to the pool of idle threads. The next time a thread is needed to run a task, if there are any threads in the pool, one of those threads is used instead of a new one. If there are no idle threads in the pool, create a new thread unless the number of threads managed by the thread pool equals a predetermined maximum. If the thread pool has already created its maximum number of threads, then tasks that need threads to run will wait until an existing thread managed by the thread pool becomes idle.

(6) *Mailbox Messages Delivery Problem*

It is not always possible to establish a connection with the intended recipient of a message at the time of messaging. Asynchronous delivery of messages is therefore desirable.

However, it is possible and acceptable for messages to be delivered without a pre-defined time frame after they have been sent.

Pattern Solution: Mailbox

A solution to the above problem has been proposed [14] and the pattern is called "The Mailbox Pattern" . This pattern provides reliable delivery of messages to objects. It facilitates the delivery of messages by storing messages to be later retrieved by each recipient.

Message sources send messages to a mailbox server along with a tag indicating the message's intended recipient. Potential message recipients poll the mailbox server for messages. When recipient objects poll for messages, they may receive more than one message. By making recipient objects receive multiple messages using a single connection, the user network bandwidth is reduced.

(7) *Transmission Messages Problem*

In transmission, messages must be reliably delivered to their recipients as soon as possible. The message sources and recipients are two parties involved in transmission. There are problems on both sides.

Messages must be reliably delivered to their remote recipients as soon as possible and, as is usually the case, they are sent at irregular intervals so that when a recipient polls for message none will be found most of the time.

Pattern Solution: Publish/Subscribe

A solution to the above problem has been proposed [92] and the pattern is called "The Publish/Subscribe Pattern" . This pattern has publisher object and subscriber object. Subscriber objects register their interest in receiving messages with publisher objects.

When a publisher object receives a message from a message source, it tries to deliver it to all of the subscriber objects that have registered with the publisher object to receive messages. Subscriber objects are responsible for receiving messages from publisher objects on behalf of receivers. Receivers tell the subscriber objects that they are interested in receiving a certain type of message.

This pattern may provide timely delivery of messages to one or more objects. Messages are delivered to subscribed recipient objects by transmitting each message to each recipient. Delivery is ensured by repeating the transmission until successful.

(8) *Remote Object Status Problem*

The amount of time that a remote operation takes to complete is highly variable, and we have no idea how long it will take.

The client needs a way of determining whether or not a remote operation is still continuing after a period of time.

Pattern Solution: Heartbeat

A solution to the above problem has been proposed [14] and the pattern is called "The Heartbeat Pattern" . This pattern allows a server to send a message periodically back to a client indicating that the remote object is still alive and performing an operation on behalf of the client.

(9) *Server's Failure in Message Delivery Problem*

Server attempts to deliver a message sometimes fail, although an object is required to reliably deliver a message to another object. This is crucial for an Object Request Broker (ORB). The only kind of call that ORB supports is a synchronous call. A reliable acknowledgement message is required in object request broker architecture.

Using a server to ensure reliable delivery by repeating delivery attempts means that it is possible for having an agent (representing a server) to deliver a message. It is particularly helpful to keep delivering a message to client by the agent even when the server (message source) has stopped running.

Pattern Solution: Retransmission

A solution to the above problem has been proposed [93] and the pattern is called "The Retransmission Pattern" . The Delivery Agent object bears the responsibility for ensuring the reliable delivery of messages. This pattern makes message delivery the responsibility of an object dedicated to message delivery rather the responsibility of the message source. To be able to deliver a message after a crash, messages waiting to be delivered must be stored on disk or other non-volatile medium.

This pattern ensures that an object can reliably send a message to a remote object. The delivery agent object designed in the pattern has the ability to handle a failure to send a message by making the object to try sending repeatedly until the message is sent successfully.

We classified the identified patterns according to the object functional natures into the Table 8.1:

Purpose	Patterns
Object Sharing	Object Identifier Pattern Protection Proxy Pattern Thread Pooling Pattern
Messaging	Heartbeat Pattern Publish / Subscribe Pattern Retransmission Pattern Mailbox Pattern
Distributed Connection	ORB Registry Pattern Server Socket Pattern

Table 8.1 Distributed Computing Patterns

Patterns related to the distributed computing issues are depicted in Figure 8.1 and the patterns will be instantiated to be meta components in MELC framework in order to support the distributed applications. As our framework is adaptable, the proposed architecture of the pattern-oriented framework has the abilities to dynamically adapt new design patterns to address issues in the domain of distributed computing as an evolutionary extension of the framework and they can be woven together to shape the framework in future.

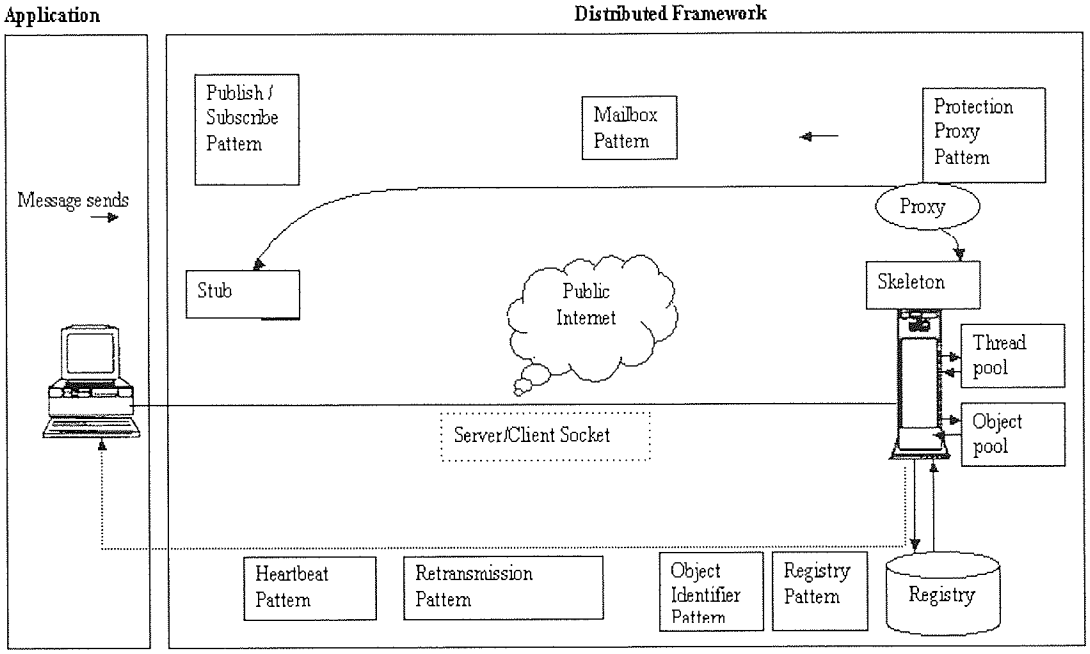

Figure 8.1 Distributed Computing Patterns in MELC Framework

The patterns for distributed computing will be instantiated as classes and registered as meta objects in the meta space. The technical details of the implementation, instantiation, and interaction of meta objects in MELC will be further discussed in the following chapters.

Chapter 9 MELC – Building Applications with MELC

9.1 MELC - A Simple E-Bookshop Application

This chapter presents the MELC framework from the point of view of an application developer, one who develops distributed computing applications. A detailed description of the development of a distributed computing application, E-Bookshop, forms the backbone of the following sections in this chapter. Although the example application given here is relatively simple, it is useful to illustrate many typical problems encountered in the development of distributed computing application. The system can be extended into a useable realistic application given further development of the application. We use this E-Bookshop to demonstrate how the MELC framework architecture helps developers solve the distributed computing problems encountered.

We follow the six steps proposed in our Adaptable Component-Based Framework Methodology described in Section 5.5 for the E-Bookshop Application.

Step One: Finding the requirements of the components in framework that may be used in the domain specific applications.

Here are the problems encountered in E-Bookshop, which are common in most distributed computing applications and have been identified in Section 8.1. We have consolidated them to produce the requirements and translate them into conditions that determine the components which meet the needs of the system.

a. *Remote Object Status Problem in E-Bookshop*: The problem identified in E-Bookshop is that the amount of time a remote operation in the

application takes is highly variable; in other words, the client has no idea how long it will take. Thus the requirement of having a component in the system provides a way of determining whether or not a remote operation is still continuing.

b. *Publishing / Subscription Messages Problem in E-Bookshop*: When delivering messages, the message sources and recipients are two parties involved in transmission. The problem identified is that messages are sent, as usually the case, at irregular intervals so that when a recipient polls for message none will be found most of the time. Thus the requirement of having a component in the system employs publisher object and subscriber object, in which the subscriber objects register their interest in receiving messages from publisher objects. So, every time, the subscriber pools for the subscribed messages sent by publishers.

c. *Server's Failure in Message Delivery Problem in E-Bookshop*: The problem identified is that, although an object is required to reliably deliver a message to another object, this is hampered by the fact that server attempts at delivery sometimes fail. Thus the requirement of having a component in the system provides that, after the message source has stopped running, the task can be relegated to an agent which makes repeated message delivery and acts as an intermediary.

d. *Mailbox Messages Delivery Problem in E-Bookshop*: The problem is that it is not always possible to establish a connection with the intended recipient of a message at the time of messaging. Asynchronous delivery of messages is desirable. Thus the requirement of having a component provides the services like mailbox which is possible and acceptable for messages to be delivered without a pre-defined time frame after they have been sent.

e. *Multi-Threads Resource Constraint Problem in E-Bookshop*: The problem identified is that, by nature, servers are presented with a steady stream of tasks that must each be performed in their own thread. Creating a thread is a relatively expensive operation, both in terms of time and memory. If too many threads are running at the same time, the overall throughput will go down. Thus the requirement of having a component in the system provides a way of optimizing number of threads that a server should be running at one time.

f. *System Evolution Problem in E-Bookshop:* The explosive growth of distributed technologies requires servers to be adaptive and configurable at run time to meet the diverse requirements of distributed computing systems.

Step Two: Selection of proper patterns in framework that fit the requirements of the application.

According to the requirements identified in E-Bookshop in Step 1, we propose, in MELC, the solutions with appropriate domain specific design patterns.

a. *Remote Object Status Problem in E-Bookshop*: In MELC, *Heartbeat pattern* [14] fits the requirements of the problem. This pattern allows a server to send a message periodically back to a client indicating that the remote object is still alive while it is performing an operation on behalf of the client.

b. *Publishing / Subscription Messages Problem in E-Bookshop*: In MELC, *Publish / Subscribe pattern* [92] fits the requirements of the problem. This pattern has publisher object and subscriber object. Subscriber objects register their interest in receiving messages from publisher objects. When a publisher object receives a message from a message source, it tries to deliver it to all of the subscriber objects that have registered with the publisher object to receive messages.

c. *Server's Failure in Message Delivery Problem in E-Bookshop*: In MELC, *Retransmission pattern* [93] fits the requirements of the problem. This pattern has the Delivery Agent object bear the responsibility for ensuring reliable delivery of messages. The pattern makes message delivery the responsibility of an object dedicated to message delivery rather than the responsibility of the message source. The delivery agent object designed in the pattern has the ability to handle a failure to send a message by making the object to try sending again until the sending is successful.

d. *Mailbox Messages Delivery Problem in E-Bookshop*: In MELC, the *Mailbox pattern* [14] fits the requirements of the problem. This pattern provides reliable delivery of messages to objects. It facilitates the delivery of messages by storing messages for later retrieval by each recipient in E-Bookshop application. A message source sends messages to a mailbox server along with a tag indicating the message's intended recipient.

e. *Multi-Threads Resource Constraint Problem in E-Bookshop*: In MELC, the *Thread Pool pattern* [95] fits the requirements of the problem. This pattern avoids the expense of creating a thread for each task by reusing threads. The threads can be managed in a way which ensures that the total number of threads never exceeds a predetermined maximum.

f. *System Evolution Problem in E-Bookshop:* In MELC, it employs a meta architecture [2] as a means of making the framework easily adaptable. The meta architecture supports dynamic adaptation of feasible design decisions in the framework design space by specifying and coordinating meta objects that represent the building blocks within the distributed system environment. The proposed meta-based framework has the adaptability that allows for the system evolution that is required in distributed computing technology.

Step Three: Creation of proprietary components in the framework. The components are instantiated with the selected patterns.

The E-Bookshop in our example is intended to provide the business services of a bookshop. We construct the proprietary components for the business services in the framework. Figure 9.1 shows that the distributed computing services like mailbox, publish/subscribe, retransmission, heartbeat, object request broker, http server and thread pool are being used to integrate and perform *Server Heartbeat Checking* (to check the distributed server running), *Web Business Application* (to provide the e-commerce services), *Message Communication Retransmission* (to retransmit messages for data transmission error because of cable accidental disconnection), *Business Ordering Functions* (to allow customers placing orders), *Sales Promotion* (to advertise sales to customers) and *Customer Mailing* (to provide mail box services to customers). In addition, administrators can use the system to maintain book stock and customer information.

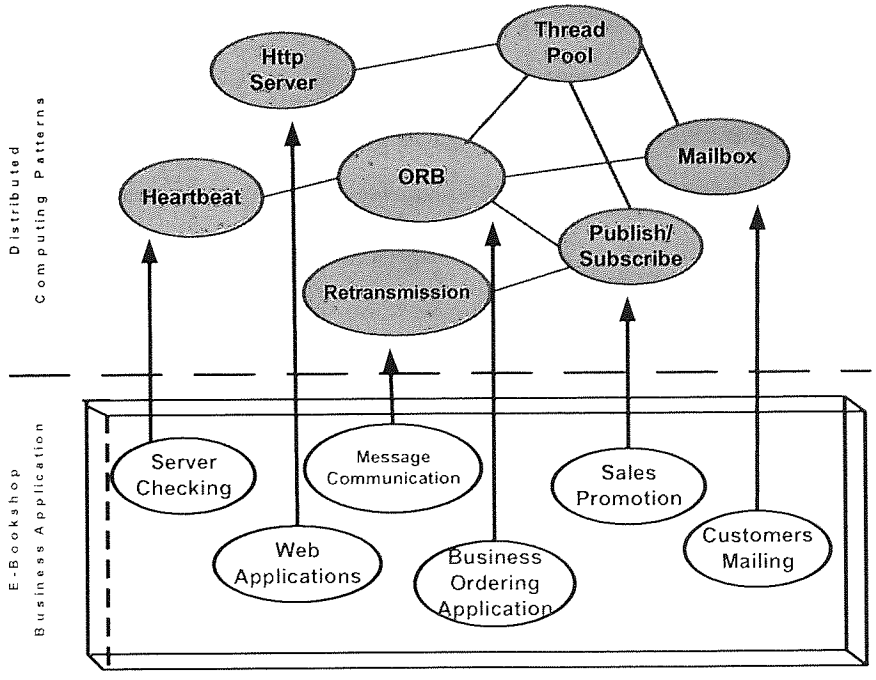

Figure 9.1 Distributed Computing Design Patterns and E-Bookshop Application

Step Four: Conformation of components to adapt to framework standards.

The selected components will be conformed so that the framework establishes environmental conditions for the component instance and regulates the interaction between component instances.

The component-based framework is a partial enforcement of architectural principles, by forcing component instances to perform certain tasks through mechanisms under control of the framework. The kernel architecture of the pattern-oriented meta-based MELC framework provides the mechanism to conform the components in MELC framework. Figure 9.2 shows the conformation of meta components from selected patterns in MELC.

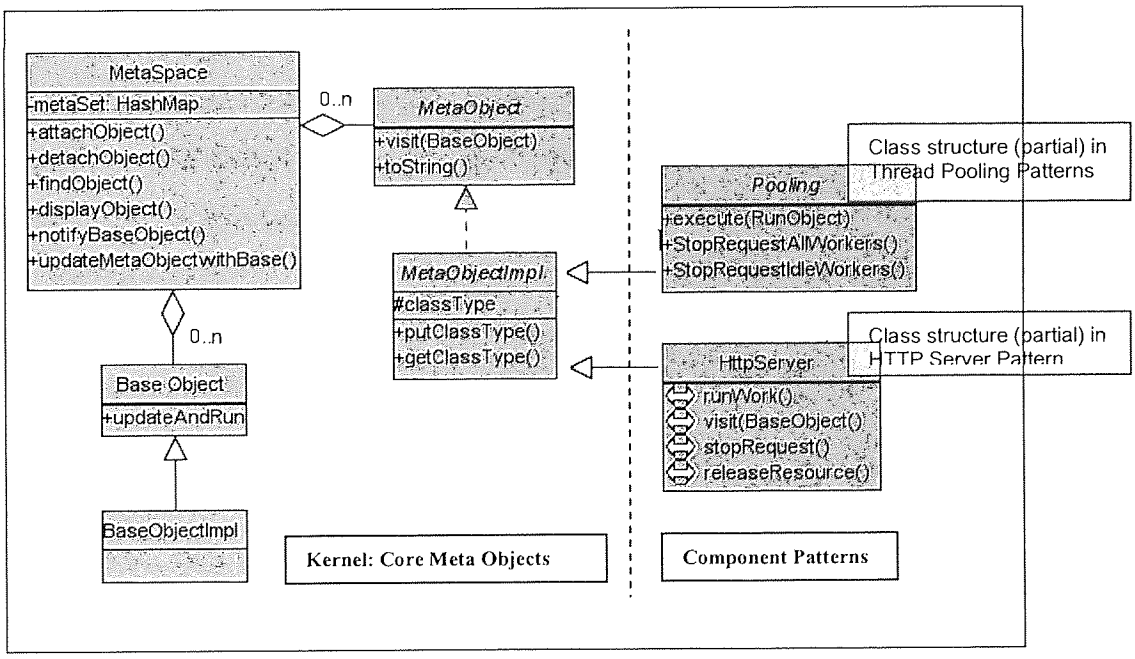

Figure 9.2 Conformation of components from selected patterns

The technical details will be further described later and here shows the simplistic way that the MELC architecture to conform meta components from selected patterns. At the class level, the class instantiation of a domain specific pattern, such as Thread Pooling patterns and HTTP Server pattern, can simply extend a class called `MetaObjectImpl` from the `kernel` package (in the left of the figure) to form a meta component, and, on the object level, once a meta component has been constructed, it is ready to deploy to meta space. The meta objects (components) in the meta space in the framework provide the system functions for the base objects (applications). We will further describe the technical details of the conformation of components for the adaptability in the framework in succeeding chapter.

Step Five: Deployment of the components.

The deployment of the components is done with a framework configuration manager. Once the components are deployed, they become part of the services in framework and are managed by configuration manager. The configuration management is the discipline that takes care of component assembly, component configuration and component integration.

MELC provides a utility called the Meta Space Kernel Manager to deploy the relevant components and to manage applications such as E-Bookshop in the framework. The utility has been developed for configuring and managing meta space. The tool allows the administrator to instantiate meta objects, and then registers them in the meta object repository.

The kernel manager has a GUI which allows the software developer control over the application they are developing. The utility allows the software developer to easily reify the distributed computing components (class instantiation of patterns) as meta objects in the application. The meta objects for distributed computing applications can be Heartbeat, Retransmission, ORB, Http Server, Mailbox, Thread Pool and Publisher/Subscriber.

Figure 9.3 shows the snapshot and procedure for the E-Bookshop Application to reify meta object, `ThreadPool`, in MELC Framework Server. The administrator uses the kernel manager firstly to select the base object E-Bookshop (Step 1) and the kernel manager will display the list of all reified meta objects list of E-Bookshop in the middle of the panel (Step 2). If Thread Pooling is not in the list and we want to include it as one of the reified meta objects for E-Bookshop application, we can select the deployed meta object `ThreadPool` in the available meta object list on the right of the panel and press the reify button (Step 3,4). The Kernel Manager will automatically include the meta object `ThreadPool` into the

list of all reified meta objects for E-Bookshop (Step 5). From now on, E-Bookshop has the thread pooling function which can efficiently provide supports for multiple tasking for the application server (E-Bookshop).

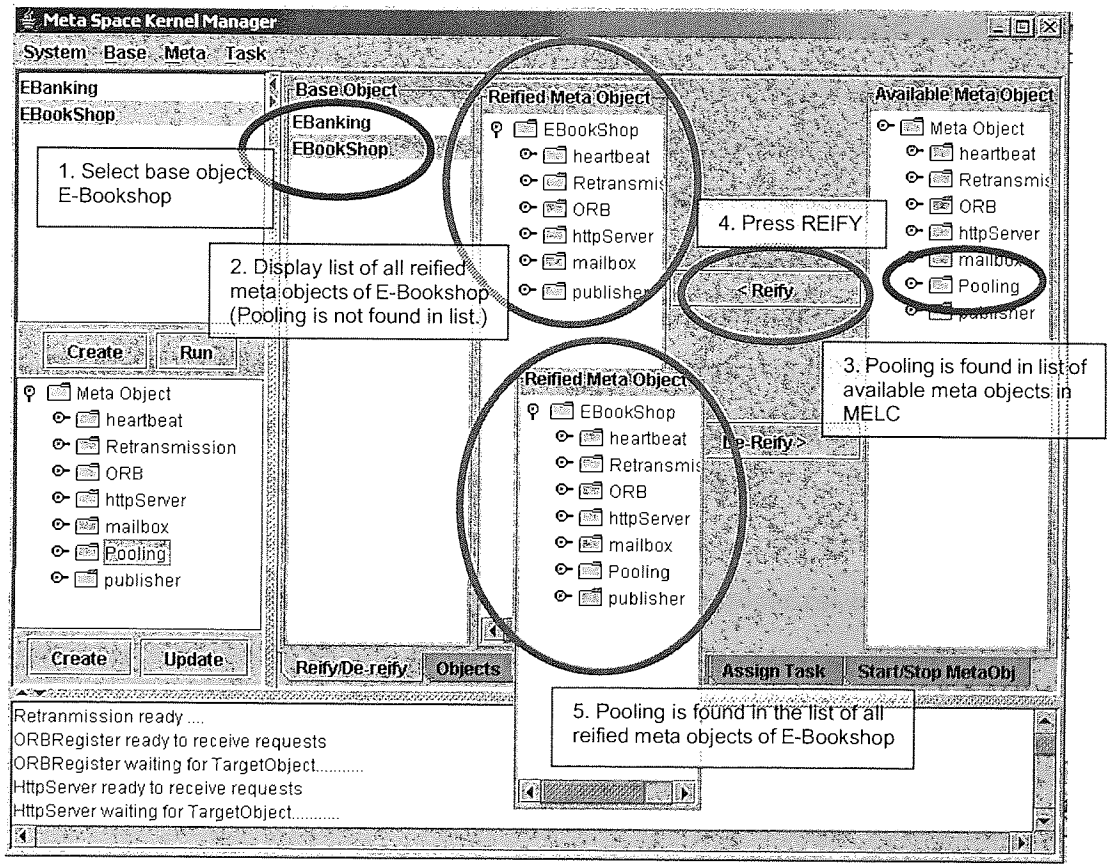

Figure 9.3 E-Bookshop reifies components in MELC Framework

Figure 9.3 shows the snapshot of Kernel Manager running with the E-Bookshop which shows all reified meta objects in E-Bookshop, such as Heartbeat, Retransmission, ORB, Http Server, Mailbox, Thread Pooling and Publisher/Subscriber. They are components (identified in Step Two of Adaptable Component-based Framework Methodology in this section) to resolve the problems encountered in most distributed applications. The application server E-Bookshop in MELC fulfills the requirements of distributed applications and

provides the essential business services of a bookshop in a distributed computing environment.

The E-Bookshop in Kernel Manager is a base object in MELC, which is an application server and provides business functions and system behavior to serve the remote clients in a distributed computing environment. The remote clients are the bookshop applications and have the graphical user interfaces (GUI) developed by the application programmers and used by application users (e.g. bookshop customers and business administrators). Each bookshop application has its own user interfaces and functions, which meet the needs of their application users.

In the object distribution environment, the bookshop applications are the remote clients. They access the remote server E-Bookshop (a base object in MELC) to invoke the meta objects in meta space to perform the functions of meta components. The seamless and transparent method invocation technology, ORB mentioned in Chapter 3, is adopted as a way of communication between remote clients and their remote server on internet platform. The ORB plays a key role in performing the remote method invocation between bookshop applications (remote clients) and bookshop server (remote server). The bookshop application (remote client) looks up the remote objects (reified meta objects) in the base object E-Bookshop via ORB and invokes the remote methods of the remote objects to carry out the system and business functions on behalf of the remote client in the application server.

The functions of the bookshop user applications (remote clients) are implemented by accessing the remote meta objects (meta components) in MELC framework.

Application developers can develop subsystems which integrate with meta

objects in the meta space in MELC. The following are the subsystems that could be implemented in bookshop application with remote services provided by the meta components:

- *Sales Promotion Subsystem* is implemented with meta objects - Publisher/Subscriber and Retransmission
- *Customers Mailing Subsystem* is implemented with meta object - Mailbox
- *Shopping Cart Ordering Subsystem, Items Searching Subsystem,* and *Item Details Maintenance Subsystem* are implemented with meta objects – ORB and Thread Pooling.

The reified meta objects of the base object E-Bookshop in Kernel Manager dynamically and transparently provide the services to remote clients in runtime environment.

Step Six: Replacing components (if applicable) as system evolves

In addition, MELC framework has the ability to provide runtime replacement of meta objects for system evolution, which adds adaptability to MELC.

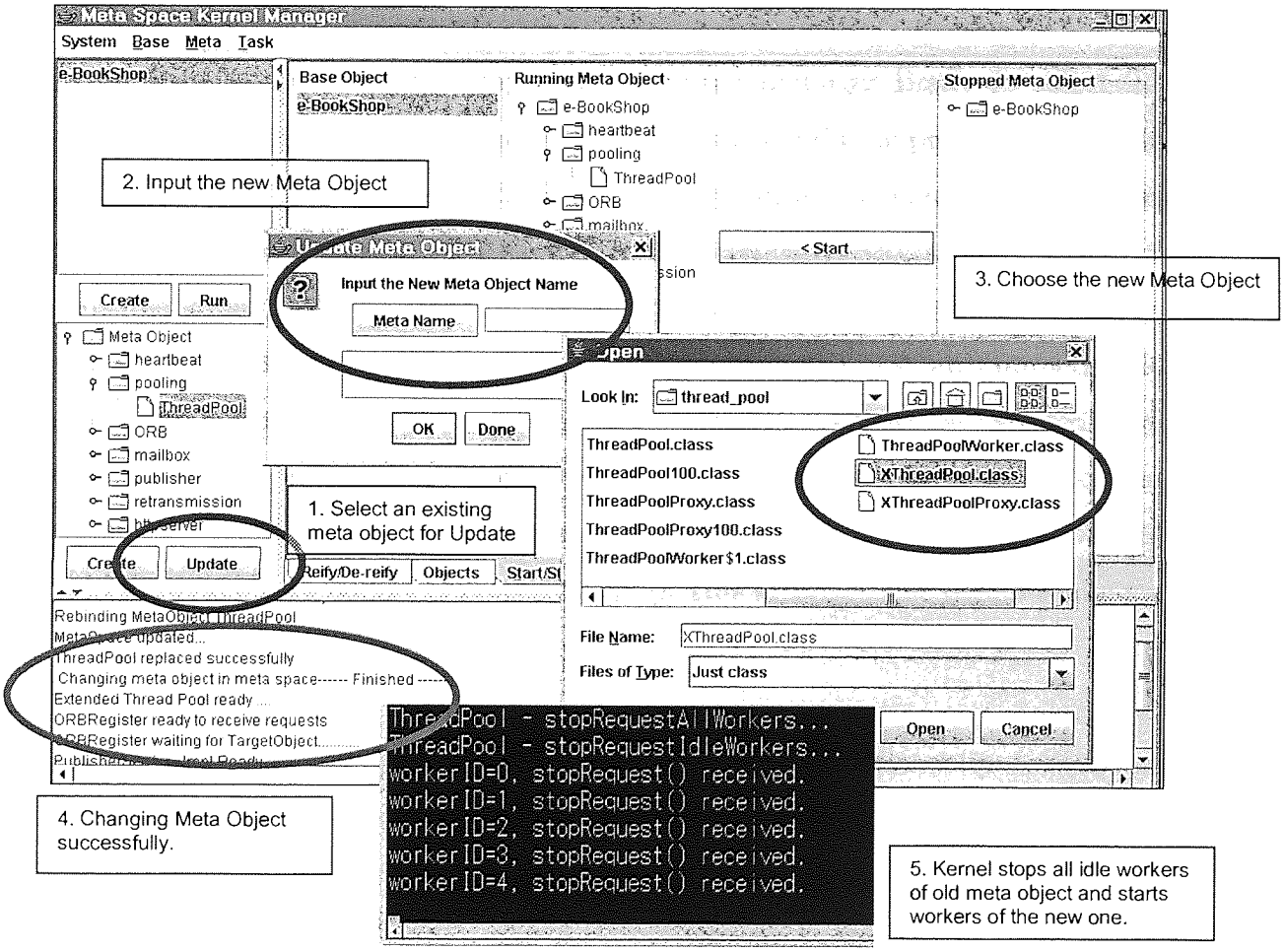

Figure 9.4 Meta Space Kernel Manager - Utility for Replacing Meta Objects

The utility, Meta Space Kernel Manager, also allows configuring and managing meta space. The tool allows the administrator to instantiate meta objects, registers them into the meta object repository, which, once created, also allows run time replacement of meta objects.

Now lets imagine the E-Bookshop needs a better implementation of thread pool and see how a new thread pool can be added without taking the application down. Figure 9.4 shows the screenshot and the procedures for replacing an existing meta component, `ThreadPool`, with the one called `XThreadPool` which is an extension of the meta object `ThreadPool`.

In Figure 9.4, the administrator uses the kernel manager firstly to select the existing meta object, `ThreadPool`, in the meta level and then presses the Update button on bottom left of the panel (Step 1). The Kernel Manager will guide the administrator to choose the new meta object `XThreadPool` stored in the file directory (Step 2, 3). During the process of replacing the meta objects, system will stop all idle workers (idle threads) of existing meta object and immediately install the new meta object (Step 4). All new threads (workers) are started from the new thread pool to continue providing the services in the system. As soon as all active threads (active workers) of existing (old) meta object finish their works, the old meta object will be removed (Step 5). This arrangement would make the replace operation quick and efficient.

The screenshot of the system console in Figure 9.4 shows the output messages while executing the `stopRequest()` method of the threads in workers of the existing (old) meta object. In the period of changeover of meta objects, it has been observed that the application server, `E-Bookshop`, is still running. Thus, the adaptability of replacing meta objects takes place at runtime. This feature is particularly important for software evolution in component-based frameworks as discussed in Chapter 2. We will further discuss the technical details of architectural design and implementation of such *adaptability* in our framework in succeeding chapters.

Chapter 10 MELC – Meta Components Installation and Integration

In MELC, domain specific patterns work as meta components [97]. In order to work as a meta component, the domain specific patterns must be conformed so that the framework establishes environmental conditions for component instance and regulates the interaction between component instances. A framework with meta components is a meta-based component framework and could be described as a set of collaborating pattern components [85, 86]. The meta-based component framework is a partial enforcement of architectural principles, by forcing component instances to perform certain tasks through mechanisms under control of the framework.

This chapter discusses the technical issues involved in developing instances of distributed computing patterns identified in Chapter 9 and shows how they can be integrated into MELC.

10.1 MELC – Meta Components Installation in Meta Space

Kernel classes in MELC are the core kernel objects. The kernel class shown in the top left corner of Figure 10.1 present the core meta classes in kernel architecture, which is used to conform distributed computing patterns to meta objects (meta components) in MELC. In this section, we use Thread Pool Pattern as an example to illustrate the implementation of meta components in meta space. The class instantiation of the *Thread Pool Pattern* [14], shown in the top right corner of the Figure 10.1, has thread pool workers defined to handle tasks assigned by the system.

The kernel architecture of the pattern-oriented meta-based MELC framework has

been described in Chapter 7. Figure 10.1 shows how the MELC architecture can provide a simple and uniform way to construct meta components.

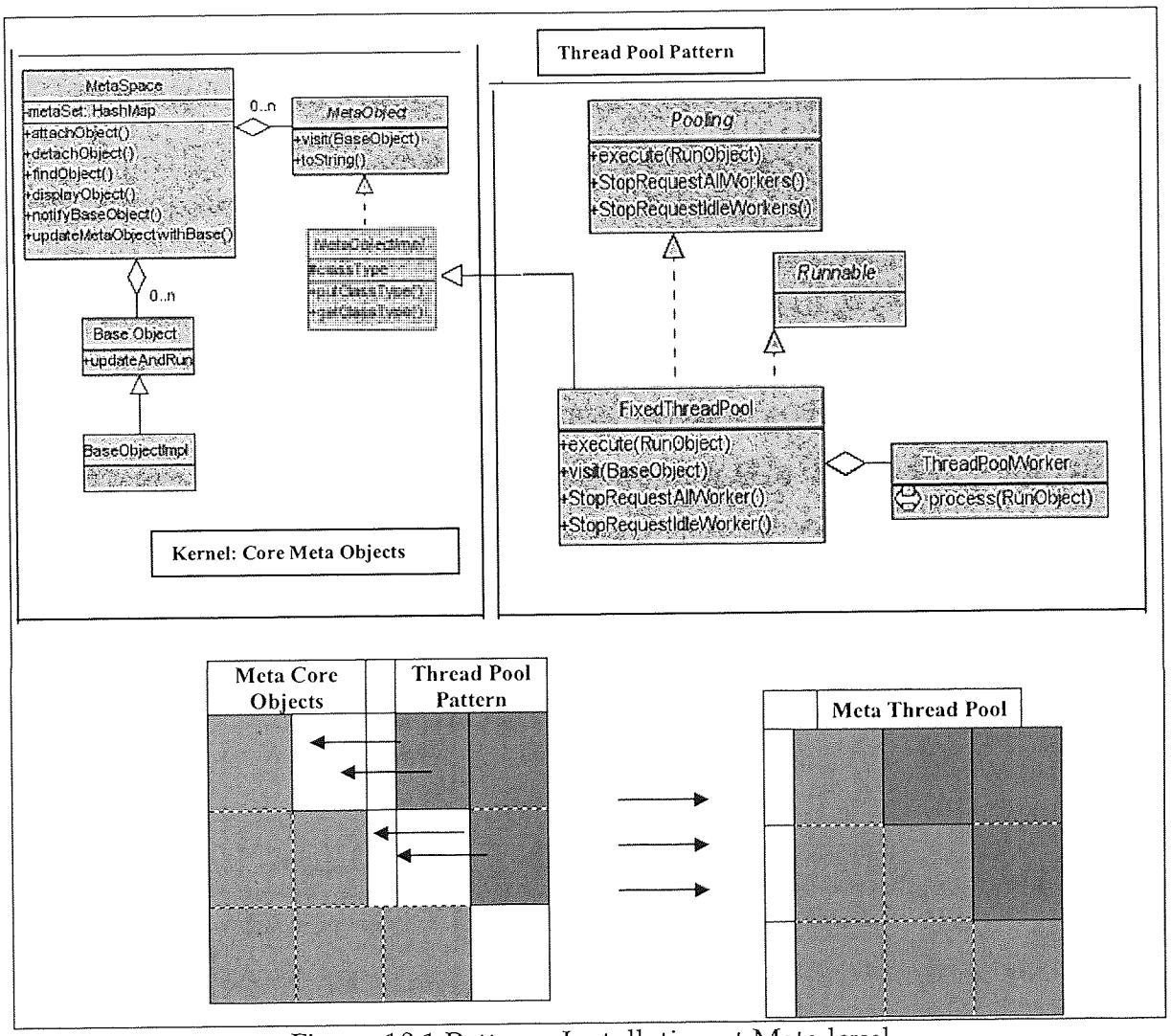

Figure 10.1 Patterns Installation at Meta level

At the class level, the class instantiation of a distributed computing pattern can simply extend a class called `MetaObjectImpl` from the `kernel` package (in the top left corner of the figure) to form a meta component, and, on the object level, once a meta components has been constructed, it is ready to deploy to meta space. The meta objects (components) of MELC provide the system functions and directly reflect the behaviors of the base objects (applications). The Kernel

Manager and its programming model described in Sections 7.4 help to manage the creation and start/stop operation of the meta objects. Thus, the meta components can be easily plugged in with MELC.

There may be similar approach to be made to conform other meta patterns to meta components into MELC. In Chapter 8, we have identified number of distributed computing patterns required to resolve problems encountered during development of distributed applications. For example our E-Bookshop example as described in Chapter 9, application requires different remote services of remote objects such as Mailbox, Publish/Subscribe, Heartbeat and Retransmission in order to implement *Mailbox Subsystem, Sales Promotion Subsystem, Server Heartbeat Checking* and *Reliable Transmission supports.* Distributed computing patterns such as Http Server, ORB Registry and Publisher/Subscriber can also be easily deployed as meta components (see Figure 10.2).

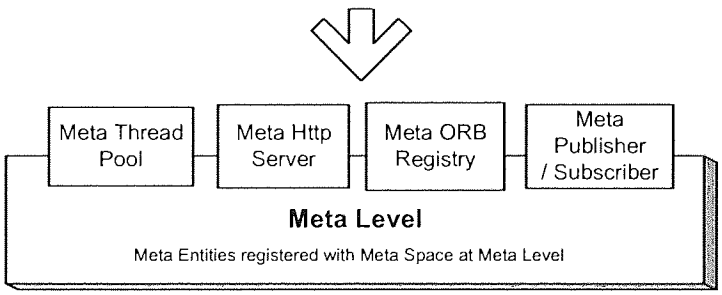

Figure 10.2 Pattern Components deployed to Meta level

After deployment, MELC consists of the technological and engineering details in the repository of meta objects. Base objects store the application-oriented details. The base level may have many application servers (i.e. E-Bookshop and E-Banking) and each application server may share technological and engineering system functions (meta components) provided at the meta level.

10.2 MELC – Meta Components Integration in Meta Space

MELC supports dynamic integration of meta objects at run-time. For example, in our E-Bookshop example, the *Thread Pooling Pattern* and *Http Server Pattern* are the two distributed computing patterns and they can be dynamically integrated together. The installation of the patterns at the class level in the way described in Section 10.1 shows the conformation of meta components in MELC. For example, in our `E-Bookshop example`, *Thread Pooling* and *Http Server* are two meta components deployed in MELC. The situation is illustrated by Figure 10.3. In Figure 10.3, *HTTP Server* can make use of HTTP Workers for the retrieval of HTML files, and *Thread Pool* can make use of Thread Pool Workers for multi-tasking processing.

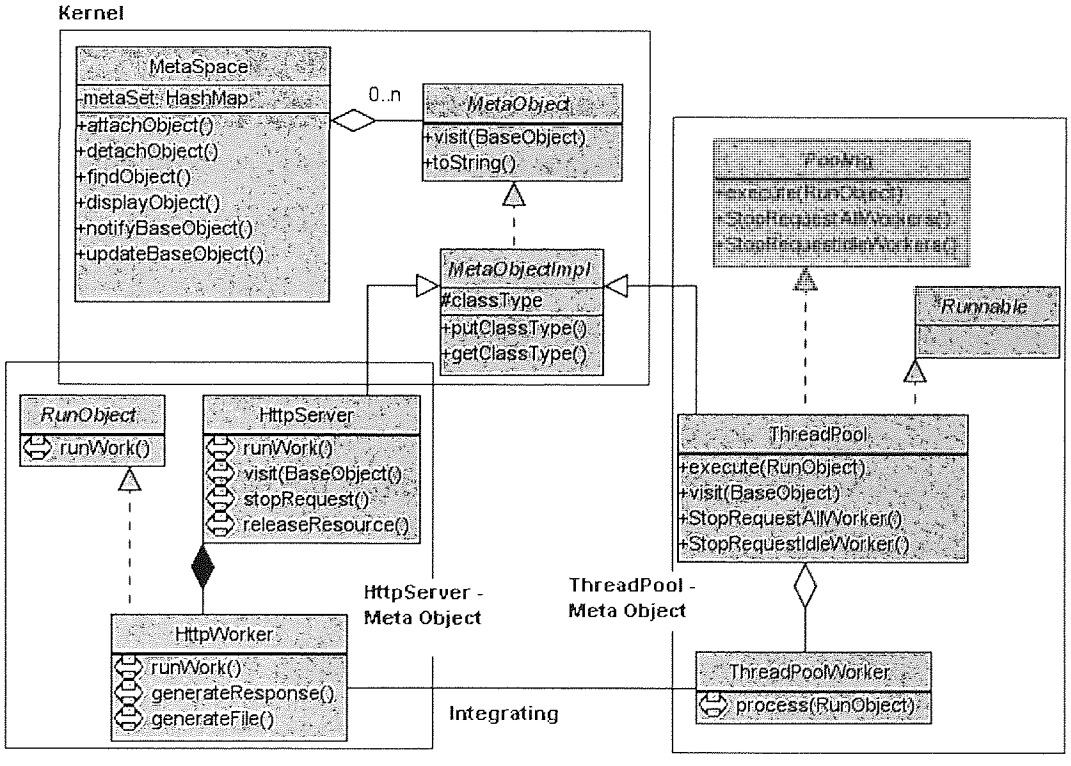

Figure 10.3 Integration happens when Thread Pool is reified by base objects

In the context of our bookshop example, the E-Bookshop base object is able to reify any number of meta objects, If the `HTTPServer` and `ThreadPool` objects are reified, it will have the functionality of both. If only the HTTPServer is reified, it will not have the `ThreadPool` functionality.

Every time a HTTP request comes to the base object, the meta space acting as a mediator in the way described in Chapter 7 will handle the request by checking whether the base object has reified the `ThreadPool`. If it has reified `ThreadPool`, `MetaSpace` will let `HTTPServer` pass its `HTTPWorker` to the thread pool workers to continue the process. The integration of meta objects, `ThreadPool` and `HttpServer` provides a *Thread Pooling HTTP Server* for our E-Bookshop application at the base level.

In our design, we have the connector object called Target Object in ThreadPool, which accepts tasks from outside and passes them to the thread pool workers to continue the process. The Target Object is the compositional connector between meta components in MELC.

The pseudo code and JAVA source code for meta component integration in MELC are presented in Section C12 in Appendix C for reference.

In the example of the Thread Pooling Http Server, the meta component name is embedded in coding, such as the word "`ThreadPool`" is hard coded in the JAVA source code for the Http Server (refer to Section C12 in Appendix C). It reduces the flexibility of the framework. In order to provide dynamic integration of components such as Thread Pool we employ the Role Based approach proposed by Tramontana [71] described in Section 5.2.4.

The Role Based approach requires a role manager which expresses the integration strategy of its components by establishing strategies that allow different components to change their integration behavior.

A role manager is associated with a meta object to generate a *Role Object* which contains the integration strategy. For example, the *RoleObject* of `HttpServer` has the name of the integration partner such as `ThreadPool`, and the *RoleObject* of `Publisher/Subscriber` has the integration partners `ThreadPool` and `Retransmission`. They depend on the strategy provided by the `MetaSpace` in the framework.

Figure 10.4 shows the collaboration diagram which illustrates use of roles in the MELC architecture for the dynamic integration of components at run time. The meta space acts as a Role Manager and contains the strategy details for component integration. The step numbers indicated in the collaboration diagram presents the sequence in processing.

In Figure 10.4, `HttpServer` requests `MetaSpace` to generate a role object (Step 1). After that, `HttpServer` asks the role object to identify the integration partners (Step 2) and checks whether it can collaborate with them in meta space (Step 3). As we mentioned before, for `HttpServer`, the integration partner could be `ThreadPool` in meta space of MELC framework.

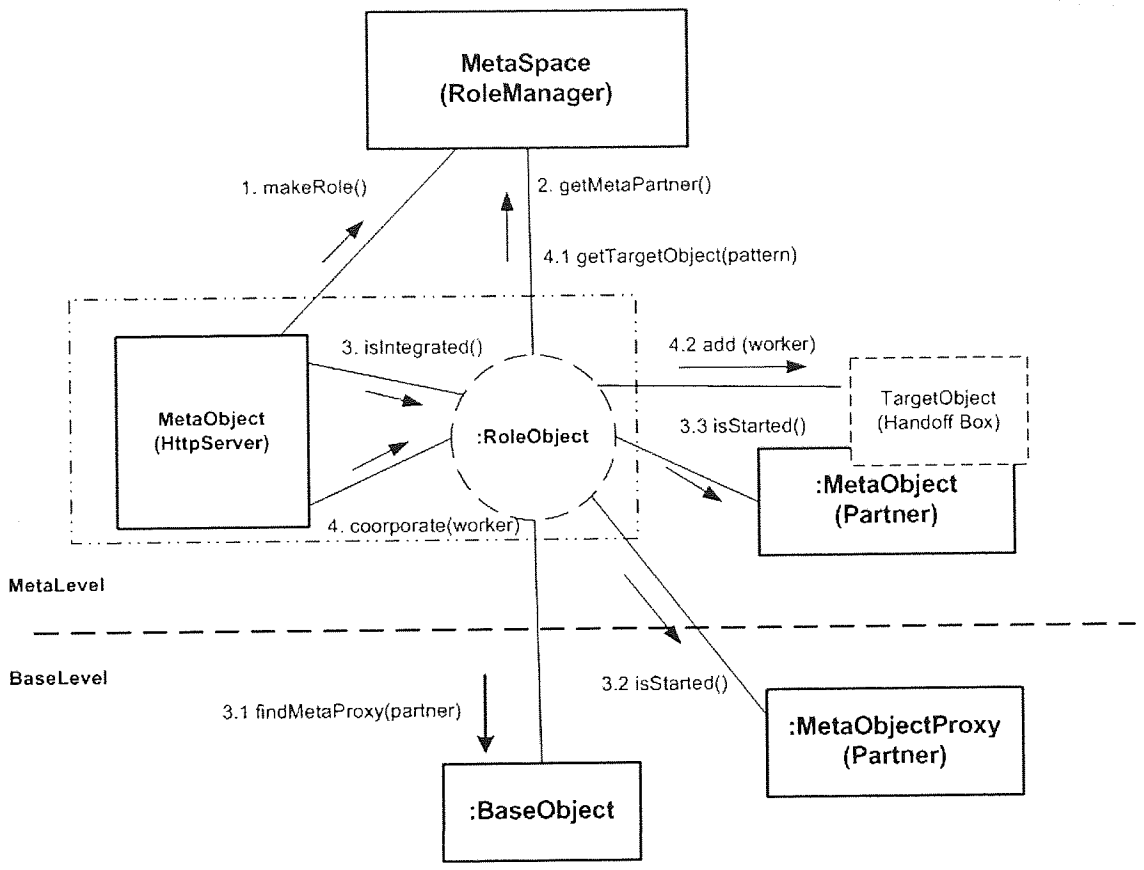

Figure 10.4 Roles deal with components integration in MELC architecture

It is important to ensure that the integration partner (such as *Thread Pooling*) in MELC has been started before the integration can take place. In this application, the role object will check with the base object `E-Bookshop` to ensure that the partner (*Thread Pooling*) has been reified and the partner's proxy object (Steps 3.1, 3.2, 3.3 in Figure 10.4) has been started and ready for collaboration with `HttpServer`. Finally, to cooperate with the partner component (Step 4), the meta object (`HttpServer`) passes one of its workers to the role object which in turn finds the connector (`TargetObject`) of its partner component (*Thread Pooling*) from the meta space repository and passes the worker (`HttpWorker`) to the connector (Steps 4.1, 4.2) so that the HTTP worker may be collected by the thread pool object.

The implementation of the role-base approach for components integration in JAVA is shown in Section C13 in Appendix C.

10.3 MELC - Meta Component Reification

This section shows the implementation of meta objects reification at the base level. We use `ORBRegister` (Object Request Broker) to illustrate how the reification happens in the `E-Bookshop` application. According to the MELC design described in Section 7.3.4, access to a meta object can be gained via its proxy and is applicable whenever there is a need for a sophisticated reference to the meta object [18].

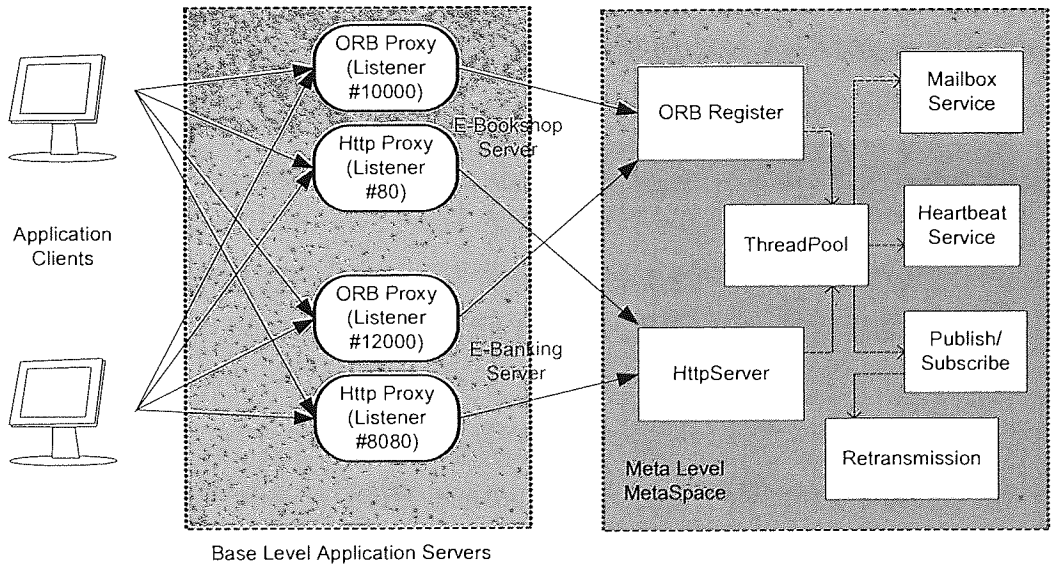

Figure 10.5 ORB Proxy and HTTP Proxy for applications at Base Level

In MELC, `MetaObjectProxy` provides a local representative at base level for each meta object that is being used by the base level application. Figure 10.5 presents the ORB Proxy and Http Server Proxy for E-Bookshop and E-Banking applications. The ORB Proxy and Http Proxy are remote meta object proxies, each having a listener with a port number and, on behalf of the meta object,

accept network connections (sockets) from remote clients. Those connections are then passed to the real meta objects, ORB Register and HTTP Server. The proxy listeners act as proxy actors and masquerade as having the functions of meta objects to accept and decide whether to send the requests to the meta level.

The sequence diagram in Figure 10.6 shows the sequence of constructing the proxy for ORB at the base level for the meta object ORB. This reification process is done using the MELC Kernel Manager.

Figure 10.7 shows a screenshot of the reification of a meta object using the MELC Kernel Manager. The screenshot shows how the software developer uses the MELC Kernel Manager to reify the meta object ORB Register for our E-Bookshop example and hence producing ORB Proxy at the base level. The developer simply selects a base object, E-Bookshop, and a meta object ORB Register with the GUI (Steps 1, 2). Once the Reify Button is clicked, the Kernel Manager begins to retrieve the `ORB Register` from the meta repository and uses it to create the meta object proxy, `ORB Proxy` (Step 3). After that, the Kernel Manager notifies the base object `E-Bookshop` that `ORB Register` is one of its reified meta objects, and from that time on `ORB Proxy` will be the ORB's local representative at the base level (Step 4).

The use of proxy objects adds *Separation of Concerns* at two levels to the framework. The operational controls can be carried out at system level (meta level) and at application level (`base level`) separately at runtime.

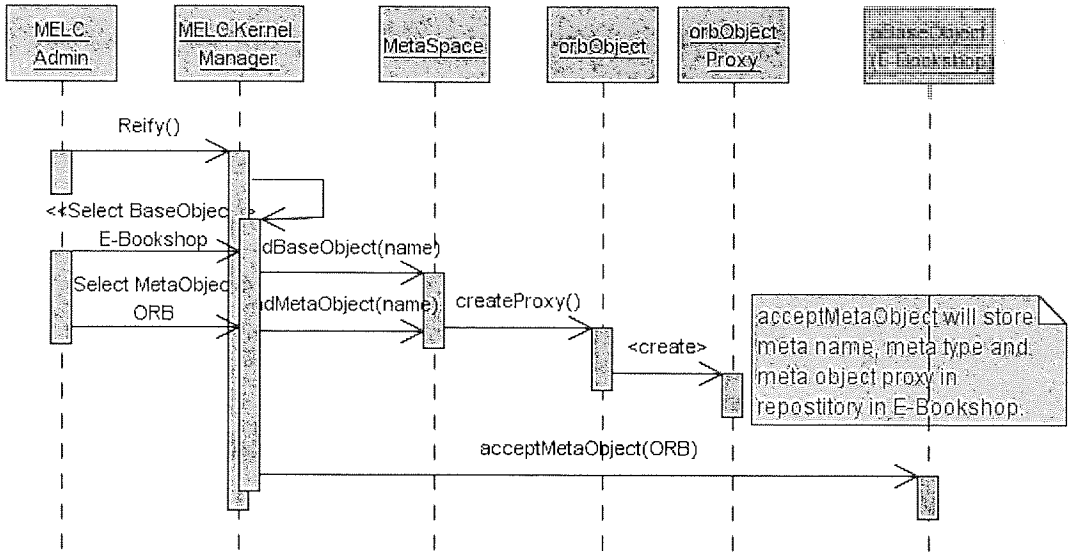

Figure 10.6 Sequence Diagram for Meta Object Reification (ORB) at Base Level

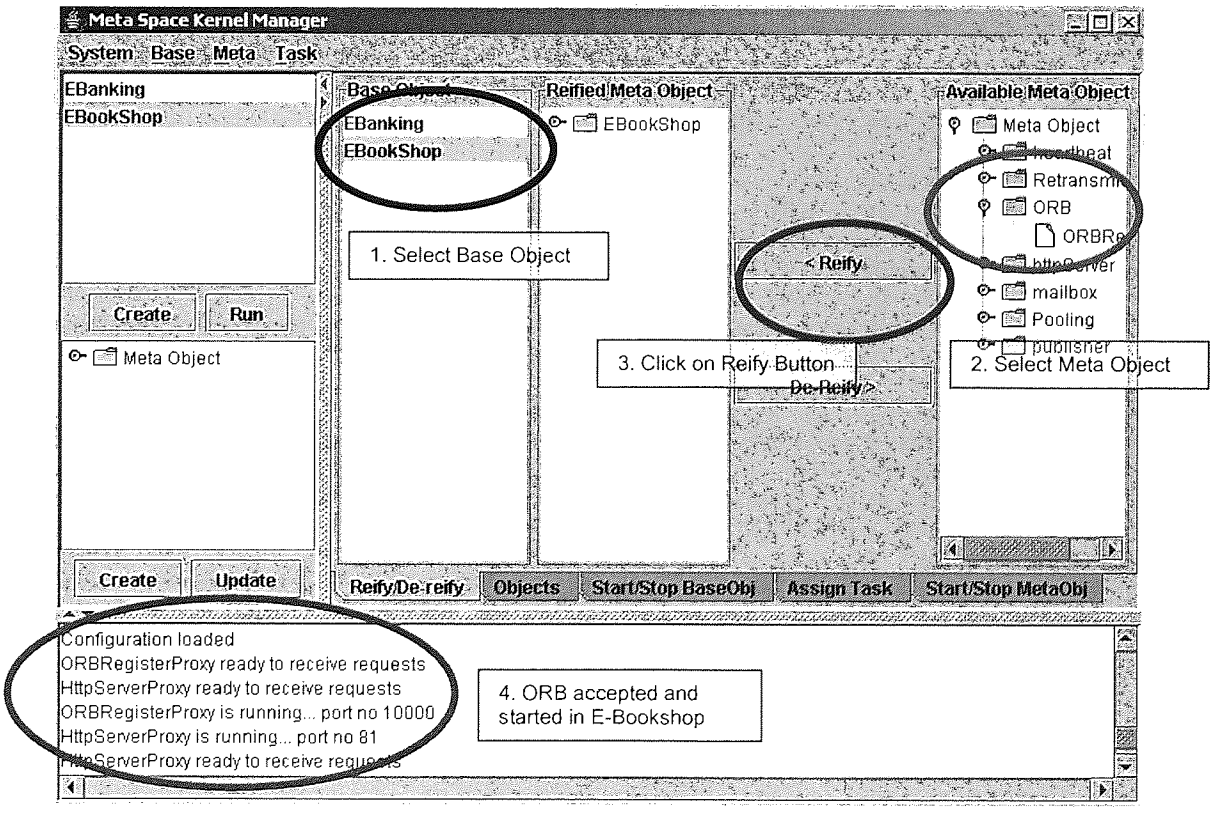

Figure 10.7 Meta Object Reification (ORB) at base level with MELC Kernel Manager

10.4 MELC – ORB Middleware for Object Distribution

Both academic and industrial researchers have agreed that Object Request Broker (ORB) is one of most common and important technologies in contemporary object distribution we mentioned in Chapter 3 and presented the technical details of Distributed Computing Technologies in Appendix D. Object Request Brokers (ORBs) are at the heart of distributed object computing and automate many tedious and error-prone distributed programming tasks [29].

An object broker is an intermediary in interactions between clients and servers. It is transparent and frees clients from having to maintain information about where a particular service is provided and how to obtain that service. It provides location transparency, so that if the server object is moved to a different location, only the object broker needs to be notified. We consider transparency in object distribution is one of important properties for distributed computing application development.

In this section, we present the class instantiation of the meta component of our Object Request Broker and describe how we use it to implement the object distribution process in MELC and add transparency and portability to MELC.

Clients of MELC in a distributed environment need to invoke methods of the remote objects in the framework. For example, the `E-Bookshop` described in Section 9.1 requires remote services of remote objects such as Mailbox, Publish/Subscribe, Heartbeat and Retransmission in MELC framework to implement Mailbox Subsystem, Sales Promotion Subsystem, Server Heartbeat Checking and Reliable Transmission supports.

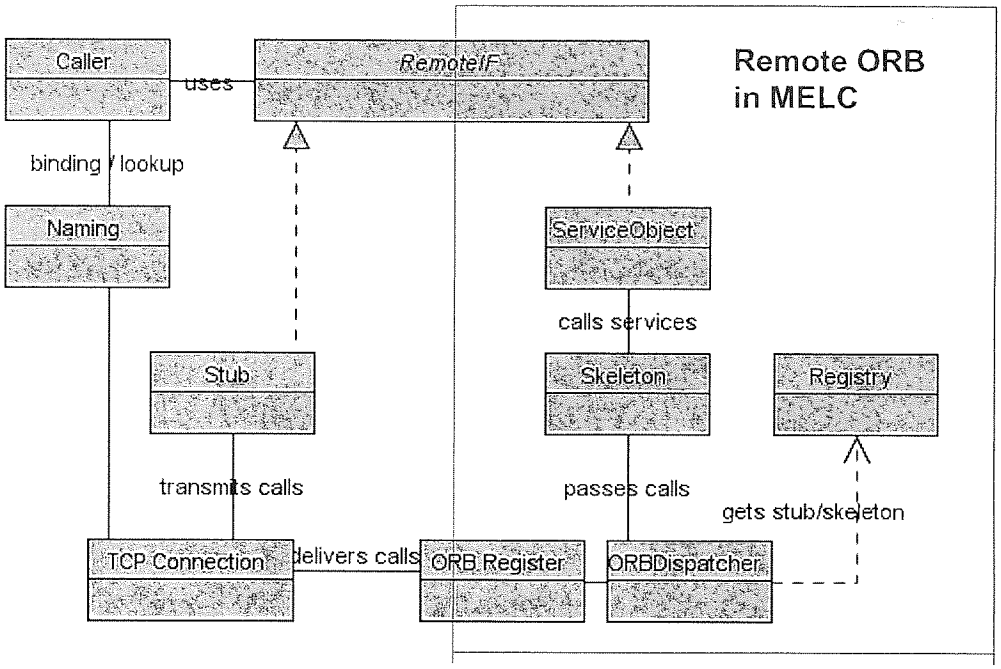

Figure 10.8 ORB Meta Component at the meta level in MELC

Figure 10.8 presents the class diagram of an ORB component [14] at the meta level in MELC. `TCPConnection` is responsible for the transport of messages between the environment of a remote caller and the environment of the callee. The `ORB Proxy` of base object `E-Bookshop` acts as a server at the base level and listens for requests. The `ORB Proxy` checks that the call has access permission and also that the required remote objects in MELC have been installed and are ready for access. `ORB Proxy` then passes the requests to the `ORBRegister` at the meta level.

In the application E-Bookshop, the callers may be the *E-Bookshop Mailbox Subsystem* or *E-Bookshop Sales Promotion Subsystem*, and the callees are the meta objects such as `Mailbox` or `Publisher/Subscriber` respectively in the MELC Framework.

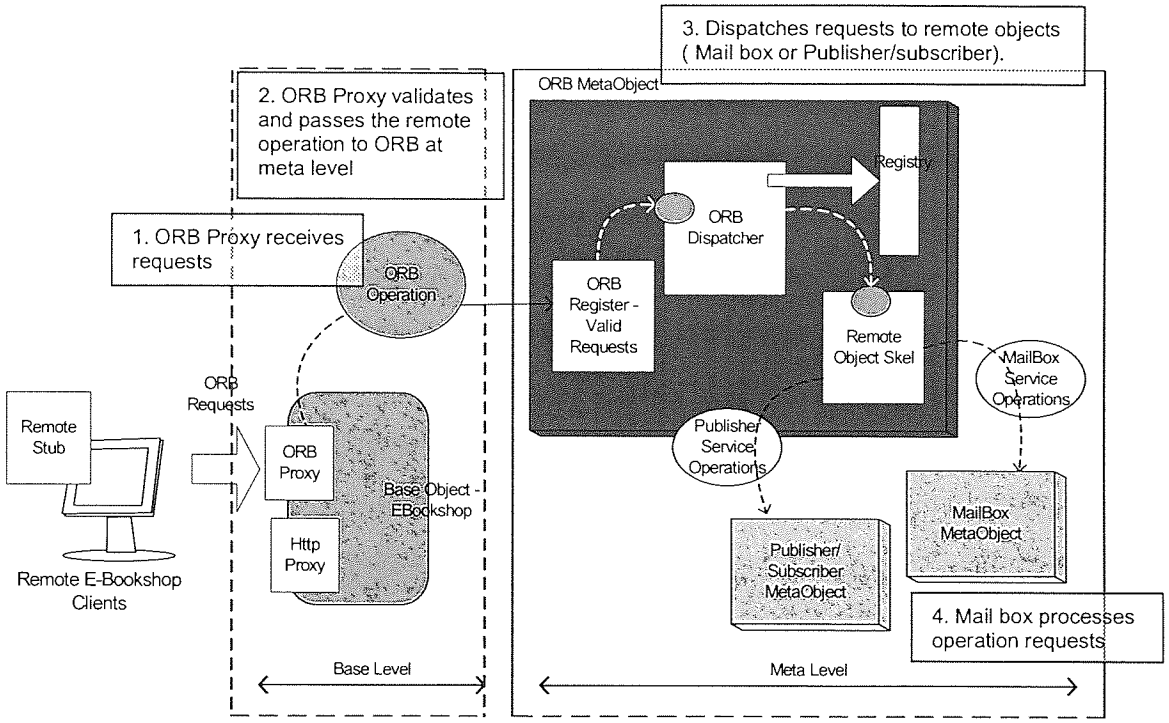

Figure 10.9 Operational flow for serving a remote request at the base level and meta level

Figure 10.9 shows the flow of remote services between the base level and the meta level in MELC. In this application, the reified meta objects in MELC, Mailbox or Publisher/Subscriber, of the base object E-Bookshop have been started and ready to be accessed. In Figure 10.9, the ORB Proxy at the base level acts as an ORB Server that receives ORB requests from the remote E-Bookshop clients, and constructs an ORB Operation (Step 1), which is passed to ORB Register at the meta level (Step 2). The ORB Register passes the requests received from remote clients to the ORB Dispatcher (Step 3). The latter depends on the invocation operation to invoke the remote services of remote objects like Mailbox Services or Publisher/Subscriber Service in MELC respectively (Step 4).

In Figure 10.9, the invocation operation, `Operation` object, is embedded in the connector, Target Object. The Target Object is a *composition connector* at the meta level, which has been discussed in Section 7.4.1. The operation object embedded in Target Object is transported to meta object ORB Register and allows the ORB Dispatcher to dispatch to the meta objects, `Mailbox` or `Publisher/Subscriber`, depending on the invocation operation for the required services processing. The design and implementation of meta component Object Request Broker in MELC adds the *transparency* of object distribution to our framework.

Note that the implementation of ORB Proxy for serving the remote requests in JAVA coding is presented in Section C14 in Appendix C for reference.

10.5 Summary

Our reflective MELC framework uses distributed computing design patterns as building blocks at the meta level. In this chapter, we illustrate the installation of patterns, the integration of components, the reification of components, and the interaction of components between meta level and base level with object distribution technology (ORB). We summarize the accomplishment in this chapter as follows:

1. With the kernel classes provided by the framework, class instantiation of a pattern can easily form meta components, which in turn can be plugged into the meta space. The construction and the reification of meta components in MELC adds *Extensibility* and *Separation of Concerns* to the framework respectively.

2. We adopt a Role Based Approach in our framework and store the integration partners in the Role Object which is generated at runtime by the role manager in meta space to support adaptation in framework. MELC provides the *Dynamic Integration* between components at meta level. This adds *flexibility* to our framework for components integration in the meta space.

3. MELC provides *Transparency* for object distribution. MELC provides object distribution middleware with Object Request Broker (ORB) (one of our meta components in meta space). The functionality of object distribution in MELC is fully transparent to users.

Hence, MELC meets the challenges of *Extensibility* and *Separation of Concerns* in framework and also provides *dynamic* integration and interaction for components. The *object distribution middleware* (ORB) for remote accesses enables the framework for distributed computing. In the next chapter, we will further describe the design and implementation of *Adaptability* in MELC.

Chapter 11 MELC - Adaptability

11.1 MELC Adaptability

The critical nature of distributed technologies requires frameworks to have adaptability which allows meta objects replaced at runtime for system evolution. In this chapter we describe technically how the design and implementation of adaptability in MELC framework. It covers the technical aspects: the *communication between meta objects*, the *design of adaptability for meta objects at runtime*, and the *implementation of adaptability for meta objects at runtime* in MELC framework.

11.2 MELC Objects Communication – Crosscutting

Cross cutting provides aspect separation in modularity [53]. The crosscutting feature in the MELC architecture provides decoupling between the base level and the meta level, which allows meta objects to be replaced at runtime for system evolution.

As we have described in Section 10.3, each *Meta Object Proxy* at the base level controls access to its meta object at the meta level in MELC. In order to support replacement of meta objects while the rest of the system stays intact, the architecture of MELC provides separation of aspects which simplifies the structure of meta components and allows objects at the base level and meta level to be described separately and exchanged independently without disturbing the modular structure of the distributed computing pattern with the framework. The *Composition Connector* [73] provides the communication connection between aspects (base level and meta level) after crosscutting in the MELC as shown in Figure 11.1.

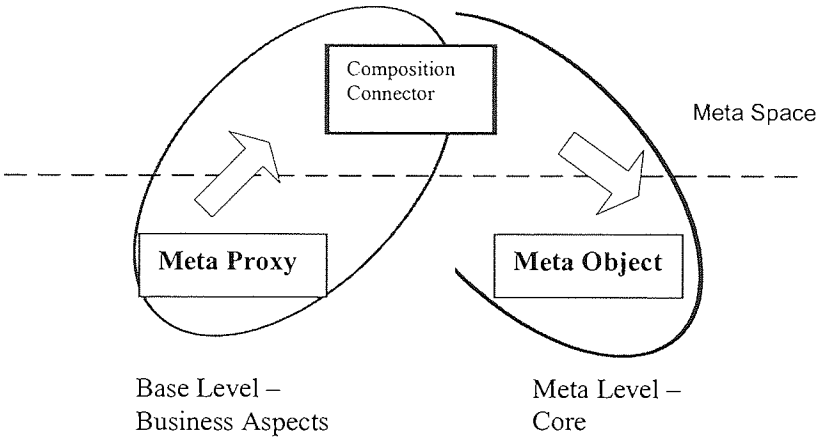

Figure 11.1 Composition Connector in crosscutting design to decouple base level and meta level in MELC

Since each meta object has a separate proxy representative at the base level, the algorithms (functional specifications) of the meta object proxy and the real meta object can be described without unnecessary entanglement. The functional requirements of *Meta Object Proxy* and its *Meta Object* in the framework are defined independently. The connection (communication link) between the *Meta Object Proxy* and the *Meta Object* we use is via the *Composition Connector* called Target Object in MELC. Composition of the crosscutting components depends on this connection. A *composition connector* allows manipulation of aspects of co-operation and co-ordination [98].

It is particularly important to note that the crosscutting feature in MELC provides decoupling between the base level and the meta level. This decoupling avoids meta object referencing at base level by *meta object proxy*, and allows replacement of the real meta objects at runtime for system evolution.

Before requests issued by remote clients reach the meta objects in MELC, they are initially received by *meta object proxy* at the base level. The meta object proxy

validates the message and checks the corresponding meta object has been started and is ready for serving before passing the message to the connector, Target Object, of the meta object. There is one target object for each meta object in meta space. The connector is a handoff box where the messages are picked up by the corresponding meta object for processing when the meta object is ready for serving.

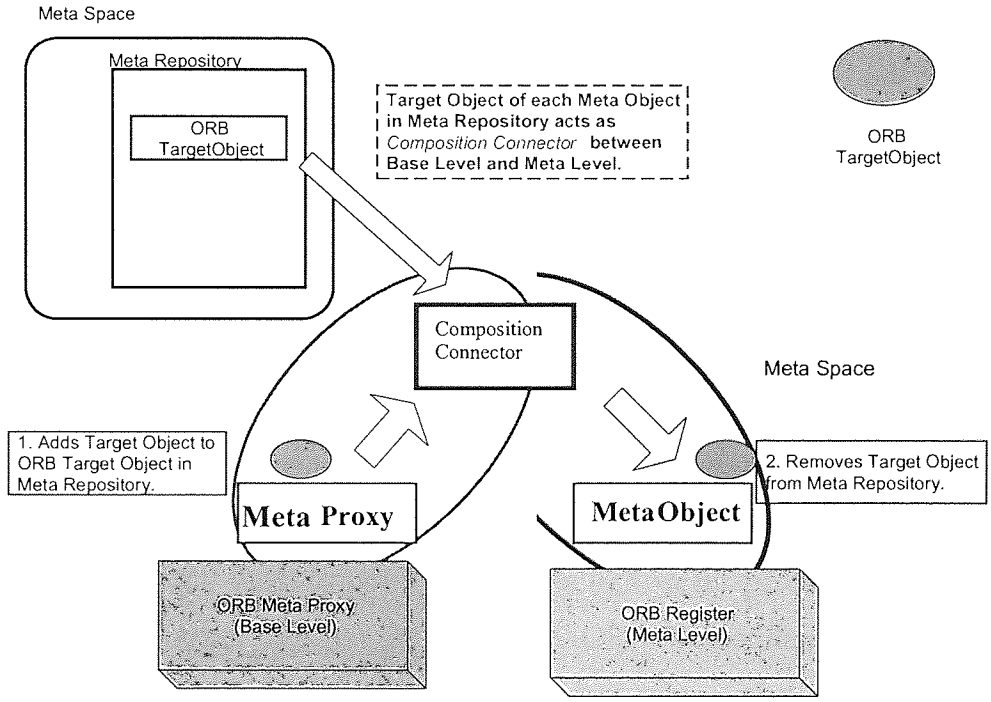

Figure 11.2 *Composition Connector* - ORB Target Object in Meta Repository

Figure 11.2 describes how the `ORB Proxy` at the base level receives the request from client. The Object Request Broker (ORB) contains an `Operation` class which is a wrapper class of the remote methods of the meta objects (such as Mailbox or Publisher/Subscriber). The instance of `Operation` class is constructed by the client, which bundles the name of the meta object and its

method that the client wants to invoke, together with its parameters, the data types of return values and exceptions. The ORB Proxy puts the `Operation` object into the ORB target object in Meta Repository (Step 1). As described in the previous section, the ORB target object in MELC architecture plays the *composition connector* role (handoff box) between meta object proxy and meta object. On the other hand, the ORB Meta Object will then remove the `Operation` from its target object (Step 2).

Note the object of the invocation operation of ORB is embedded in the connector (Target Object). Whenever ORB Meta Proxy puts the `Operation` object to ORB target object in Meta Repository, the ORB Meta Object immediately removes its target object from Meta Repository in the meta space and processes the embedded `Operation`, and ORB meta object dispatches it to the relevant meta object (such as Mailbox or Publisher/Subscriber) for processing.

In the meta space, the ORB meta object locates and retrieves its connector, target object, from the meta repository in meta space. Inside the target object, the ORB meta object can extract the information of the remote request: the base object, the request's socket, and most important the `operation` for processing. After that, the ORB meta object assigns its work to ORB Dispatcher to process the `operation` obtained from the target object, where the ORB Dispatcher dispatches the `operation` to one of the meta (system) components like *Mailbox* or *Publisher/Subscriber* in `E-Bookshop` for processing.

The detailed JAVA codes for the ORB meta object to dispatch the remote requests at meta level have been extracted in Section C15 in Appendix C for reference.

11.3 MELC - Design for Adaptability of Meta Objects at Runtime

MELC architecture is designed to adapt to new user requirements by replacing software components at run time while the rest of the system is still running. In this section, we describe the design of MELC to adapt to changing requirements. For example, if MELC contained a component implementing a basic thread pool with a fixed number of threads which will call a *Fixed-threaded Thread Pool* to support multi-tasking processing in a server, through system evolution, that component could be replaced in MELC by a thread pool with a variable number of threads which we will call a *Growth Enabled Thread Pool* [99].

It has been observed that the advantages of using the crosscutting feature and the composition connector (`Target Object`) described in the previous section can avoid the cohesive dependency between components, and that mechanism facilitates *adaptability* in the MELC framework. As discussed in Section 7.4, we proposed that each meta object should have its own internal identifier which we call meta object identifier and which is private in meta object. The Fixed Thread Pool Pattern and the Growth Enabled Thread Pool Pattern are two typical examples of pooling patterns [14]. The class instantiation of both are depicted in Figure 11.3.

Each meta object used in implementing these two patterns will have its own internal identifier. However, for the two thread pool patterns, the contents of the two meta object identifiers are identical, which indicate that they both provide the same functionality and they are interchangeable. The meta object identifier is set by the meta component designer and it can be changed or overridden by its subclasses whenever necessary. However, non-identical meta object identifiers in meta objects represent different meta components in MELC and they can not be interchangeable for each other. The meta object identifiers of meta objects will be verified while replacing in MELC at runtime.

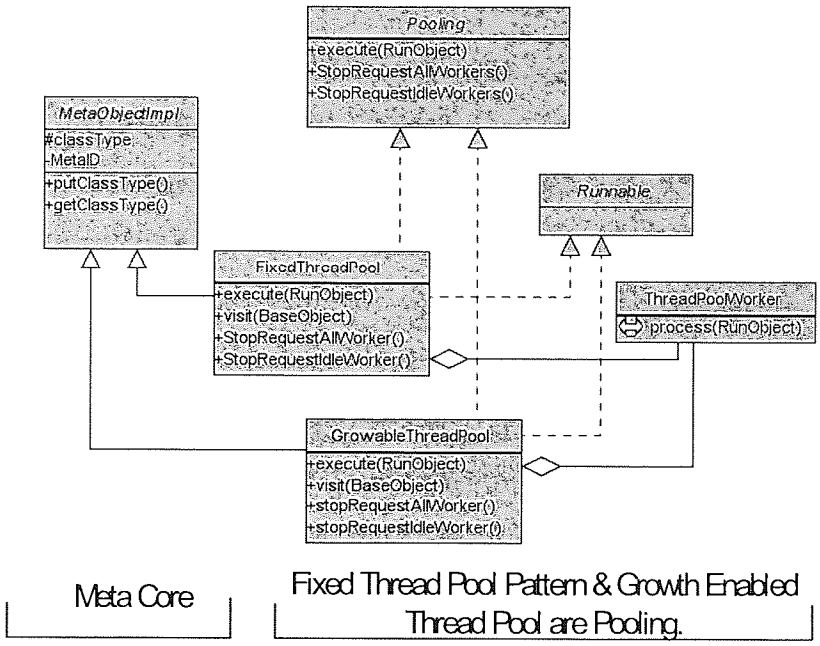

Figure 11.3 Meta Objects: Fixed Thread Pool and Growth Enabled Thread Pool

MELC Kernel Manager has the ability to replace meta objects at runtime by verifying their internal identifiers. One meta object can replace another if they are of the same type and have the same internal identifier. Therefore a fixed thread pool could be replaced with a growth enabled thread pool as described below. Figure 11.4 illustrates the adaptability in meta objects.

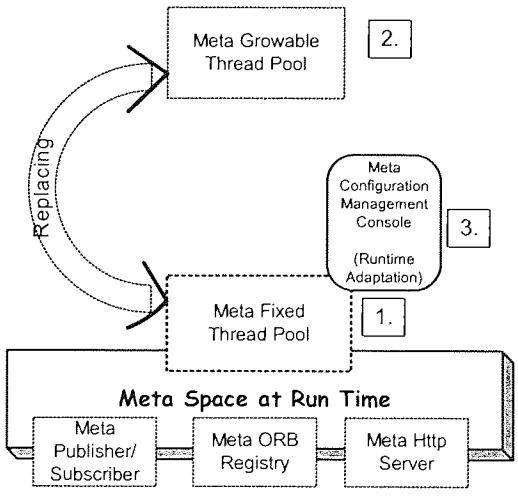

Figure 11.4 Adaptability of Meta Objects in MELC

The procedure for replacing meta objects is listed as follows:

1. `MetaSpace` has constructed and instantiated `metaFixedThreadPool` which, while belonging to a Type called `Pooling`, possesses an internal identifier with the value `ThreadPool`. The meta object, `metaFixedThreadPool`, has been registered in the meta object repository.

2. Meta object `metaGrowableThreadPool` is constructed and instantiated and since it belongs to a type called `Pooling`, it also has an internal identifier with the value `ThreadPool`. However, it has not been registered in the meta object repository. Since it has the same meta type and internal identifier as `metaFixedThreadPool`, it can each be replaced by the other.

3. Our configuration and management utility controls the replacement of meta objects. The changes will reflect the system behavior at the meta level and also immediately affect those base objects which have reified the meta objects possessing the internal meta identifier `ThreadPool`.

11.4 MELC Implementation for Adaptability at Runtime

The key challenge in MELC is to adapt to new user requirements by replacing software components concerned at runtime while the rest of the system is still running.

In MELC, after meta objects have been constructed and reified by base objects, they can be replaced at run time as system evolves. The process of replacing meta objects in framework is shown in the collaboration diagram in Figure 11.5.

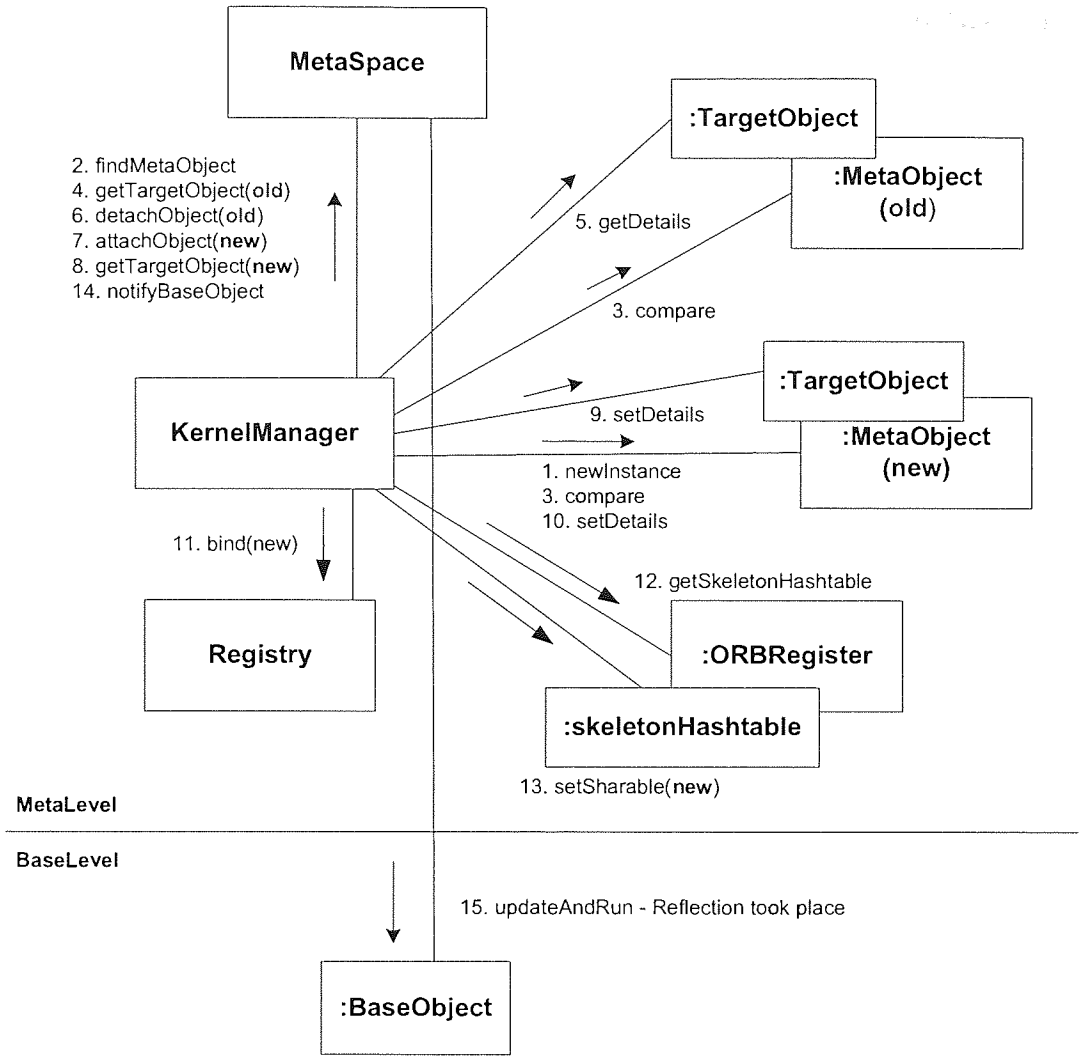

Figure 11.5 Collaboration diagram for meta objects replacement at runtime

Figure 11.5 shows how the replacement of meta objects in MELC take places at runtime. The procedure can be described as follows:

1. Kernel Manager first instantiates the *new* meta object by using a given meta object name and the *new* meta object is then stored in meta objects repository in meta space. At the same time, Kernel Manager constructs the Target Object of the meta object, which is the connector (a handoff box) mentioned in Sections 11.2 and 11.3 when the meta object is stored in the meta objects repository. Using the

Target Object in meta space is to avoid the cohesive dependency between meta objects. (Step 1)

2. Kernel Manager then verifies existence of the *old* meta object in meta space and uses the meta object type and meta object identifier of the two meta objects to compare and ensure that the two meta objects are identical. For example, the meta objects, `metaFixedThreadPool` and `metaGrowthThreadPool`, have the same identity `ThreadPool` and belong to the same meta type `Pooling`. (Steps 2, 3).

3. Before the *new* meta object replaces the *old* meta object, the Kernel Manager transfers the information in the target object such as system table or port no of the old meta object to the target object of the new meta object. (Steps 4, 5, 8, 9, 10).

4. In between the replacement of meta objects, Kernel Manager removes the *old* meta object from the meta repository and releases all its resources, and stores the *new* meta object in the meta repository and claims all necessary resources if applicable. (Steps 6, 7).

5. For object distribution, Kernel Manager rebinds the *new* meta object in the registry in the Object Request Broker. (Step 11).

6. The registry in the Object Request Broker (ORB) has a hash table which contains all skeletons of meta objects in meta space. A Skeleton in ORB is responsible for calling methods of *Callee* objects (meta objects in MELC) on behalf of remote clients. Kernel Manager refreshes the skeleton hash table in the registry with the skeleton of *new* meta object. (Steps 12, 13).

7. At the end, Kernel Manager notifies the base objects that have reified the

replaced (old) meta object to accept the new meta object for performing object reflection in the architecture. (Steps 14, 15).

The detail Java coding for the implementation of replacing meta objects at runtime has been extracted in Section C16 in Appendix C for reference.

Kernel Manager instantiates a new meta object with the method `newInstance()`, where the variable `newClassName` stores the class name of the new meta object. For example, the new meta object *Growth Enabled Thread Pool* has the Java class name `metaGrowableThreadPool.class` to replace the old meta object *Fixed Thread Pool* which has the Java class name `metaFixedThreadPool.class`. To meet the requirements of meta objects replacement, they both must have same meta object type and meta object identifier. In this case, they both are `Pooling` and `ThreadPool` respectively. The old meta object is removed from meta repository before the new one stored.

The object references in meta objects are instances such as the skeleton table stored in the ORB meta object and the port numbers used in meta objects. They are called Concerned Instances in distributed environment [82]. Note a skeleton in ORB is responsible for calling methods of *Callee* objects (meta objects in MELC) on behalf of remote clients.

In MELC, the skeleton table and port number are the concerned instances of the old meta objects and the new ones in the process of replacement in distributed computing. While replacing the meta objects, the concerned instances of the old meta object are transferred to the new one. As described in the Section 6.5 of Chapter 6, it has been noted by academic and industrial researchers [82] that distributed enterprise application servers like J2EE and JBoss, the *concerned instances* in EJB objects (caches, cookies, session objects and session beans) can

not be passed to the new component as system evolves. As a consequence of version barrier of current application servers, it is difficult to perform adaptation at runtime with distributed servers. That is to say, the current distributed application servers (J2EE) can not provide *Hot Evolution* [82].

Our MELC framework takes care of the *concerned instances* of meta components while replacing them at runtime, which allows MELC to be adaptable as system evolves. It means MELC supports hot evolution to distributed servers.

11.5 Summary

Both academic and industrial researchers are aware of the importance of having component adaptation for distributed object-oriented enterprise framework [81]. In order to achieve this, software designs must provide flexible architectures that can more quickly adapt to changing requirements. In an attempt to address the concerns of adaptation, the reflective models and frameworks described in Chapters 5 and 6 have designed and implemented reflective architectures (meta architecture), although they failed to address the problem of system evolution in the runtime environment.

The design of MELC uses *Composition Connector* in *Crosscutting* to support adaptability for the meta components and allow object distribution with the middleware of Object Request Broker (ORB).

This chapter technically illustrates the design and implementation of adaptability in distributed computing environment. The implementation shows how MELC replaces meta objects at runtime and resolves the problem of concerned instances encountered in enterprise servers in dynamic adaptation at runtime. We summarize the accomplishment in this chapter as follows:

1. The design of *composition connector* avoids object referencing between the base level and the meta level. With the *Composition Connector* in *Crosscutting*, the remote requests can be received and handled by `ORBRegister` (Object Request Broker), which in turn dispatches the requests to the appropriate meta components for processing at the meta level.

2. MELC framework is adaptable at runtime. The architecture has the abilities to dynamically replace meta components for mission critical applications for software evolution or adapt new design patterns to address issues in the domain

of distributed computing. They can be woven together to shape the framework in future.

3. It is important that *concerned instance* be passed between the old and new versions of the component. Such *concerned instances* in a lightweight component model of ORB architecture are caches and skeleton tables. MELC resolves the problem of concerned instances encountered in recent distributed enterprise servers.

In Table 11.1, we summarize the accomplishment and apply the same assessment attributes used in Chapters 5 and 6 to the MELC Framework.

Thus far the techniques involved in developing MELC framework have emerged as a promising way to meet current and future challenges in *adaptability* in the distributed environment.

Adaptable Frameworks	MELC [85,86]
Brief Description	Meta level Component-based Framework with pattern-oriented approach to provide adaptability for runtime replacement of meta objects at meta level. Domain specific patterns are used as building blocks, and are instantiated to be meta objects in framework. Those meta objects are meta components of MELC. Patterns are meta components that provide high software quality. Meta components in MELC can be easily installed into the framework in a uniform way and dynamically integrated with each other in the framework. MELC has online utility for managing meta components in framework.
Meta Objects Coordination (meta space)	Meta space is a mediator to coordinate base objects and meta objects. It has the repository for base objects and meta objects. It is the kernel of the framework.
Meta Objects Integration	Role manager in framework generates a role for each meta object. Role manager checks with existence of meta objects in meta space. Role consists of information and performs dynamic objects integration.
Meta Object Adaptation	Dynamic object adaptation (objects adding, changing and replacing) for system evolution at run time.
Communication linkage between Meta Level and Base Level	Causal connection is used in framework. Meta proxy at base level intercepts the method invocation and connector helps to build the interface between base level and meta level and passes the requests between two levels.
Configuration Management	MELC has Kernel Manager for resources and configuration management.
Evaluation on properties of adaptable framework	Meta objects are the instantiation of distributed computing patterns are the system components and they can be easily adapted at runtime. The architecture of MELC is platform independence and language independence. It performs run time adaptation for meta objects at meta level for system evolution for distributed computing applications or applications of critical missions which do not allow offline for system upgrades.
Object Distribution Middleware	ORB is one of meta components and conducts the function of object request broker (ORB) in framework for distributed computing applications.

Table 11.1 Adaptable MELC Framework for System Evolution

Chapter 12 MELC Performance Evaluation

12.1 MELC Performance Evaluation

The unrelenting pace of change in distributed computing that confronts contemporary software developers compels them to make their applications more configurable, flexible, and adaptable. In this thesis, we have proposed a Meta Level Component-Based Framework (MELC) which uses distributed computing design patterns as components to develop an adaptable pattern-oriented framework for distributed computing applications.

While distributed component-based technologies provide an infrastructure solution for distributed computing applications, it is difficult to accurately measure the effect of the framework implementation on eventual performance of an application built using that framework. In fact, component technologies (DCOM, J2EE and JBoss) make the problem even more difficult, as each component technology may have a different infrastructure and implementation, and thus exhibit different performance characteristics [100]. Considerable work on software performance modeling has been done in the academic community. However, most existing models are either too complex to use or do not take the underlying infrastructure into account, making them impractical for performance prediction of component-based applications in industrial software engineering projects.

Recognizing these issues, in this thesis we investigate the feasibility of our proposed solution by evaluating the performance of MELC. An empirical approach is proposed to determine the performance characteristics of MELC framework. A benchmark is used to exercise components and measure their

behavior. Deeper observations on meta components and application components in MELC have been conducted from these empirical results.

12.2 Test Suite Design

In order to obtain a performance profile solely underlying MELC framework and minimize the effects of application behavior, an application of an E-Bookshop, has been designed and implemented as our basic benchmark (see Chapter 9). The benchmark has several important characteristics that make it appropriate to use for examining and evaluating the costs of implementing an application within the MELC framework in comparison with an application which employs distributed middleware Object Request Broker (ORB) for distributing computing. For comparison purposes, two versions of the e-bookshop application have been implemented. One was implemented using the MELC framework and the other using traditional ORB technology. Both versions of the e-bookshop application have:

- exactly the same distributed computing services (system components): ORB Registry, Heartbeat, Mailbox, Publisher/Subscribe, Retransmission and HttpServer.
- exactly the same user interfaces and business functions (application components) for administration and sales: Customer/ Book/ CD Searching, Shopping Cart, Order Maintenance, Items Maintenance, Mailbox Service, Subscription, and etc.
- the same testing sample data in databases

The key differences between two applications are:

- One is an adaptable component-based application (E-Bookshop integrated with MELC Framework) implemented with reflective meta-architecture.
- The other is a non-adaptable component-based application (E-Bookshop

without MELC Framework) and is non-reflective for runtime adaptability

The test focuses on the capturing of the response times of the system components (system services) and application components (business services) in the application server. The transmission time for data communications via networking with sockets is not taken into account because we want the data captured to reflect the performance of the components in the server and not be influenced by the speed of the network communications. The measurements we have taken are measuring the overhead caused by the reflective meta-architecture for providing the adaptability to the system.

12.3 Benchmarking

In distributed object broker communication mentioned in Chapter 3, two key performance measures are:

1. Response time for client to *look up remote brokers* resided in distributed server;
2. Response time for distributed server to *execute the remote requests* from client.

As an example of the effectiveness of the benchmark application, Figure 12.1 shows the average response time (ART) breakdown for *looking up* business components (Customer Service, Book Service, Ordering Service and etc) through ORB infrastructure. The test platform comprised Windows XP machines. A 100 Mbps network connected the machines. A JAVA profiling tool, Performance Measurer, was developed by the author for this project and used to obtain the performance metrics, such as response time of components and the number of invocations. Because the components under test are treated as black boxes in borh frameworks, whenever a request calls a component, the operations of the associated objects involved are hidden. We capture the runtime response time

and aggregate them to find the average response time (ART).

To efficiently support large numbers of simultaneous access from clients, multi-threaded servers must be used to increase the processing capacity of an application. Thread pooling is one of essential services in most of distributed application servers [100]. MELC provides the capability for adapting the Thread Pool component by plugging it in the framework, which creates a pool of threads and can dynamically integrate with components in the server to perform a multi-threaded distributed computing server to handle requests concurrently.

We assume the concurrency is an indispensable feature and investigate how much is the difference in performance in application with MELC framework in comparison with traditional component-based approach. The benchmark application, E-Bookshop, is evaluated through three different approaches under different infrastructures:

- A component-based development (CBD) approach which does not use MELC but has ORB infrastructure for object distribution (non adaptable application);
- An adaptable component-based approach which uses MELC in which ORB component is reified for object distribution;
- An adaptable component-based approach which uses MELC and in which ORB and thread pool components are reified for object distribution and concurrent accesses respectively.

These three approaches have the same distributed computing services (same system services), same implementation for user interfaces and business functions for administration and sales for E-bookshop, and same testing sample data in database. Technically, the key difference in implementation is that the adaptable

approach has system components (distributed computing components) integrated with reflective MELC framework for providing the adaptability to the components at runtime. However, the non-adaptable approach has system components integrated with applications directly and the components can not be replaced for system evolution at runtime.

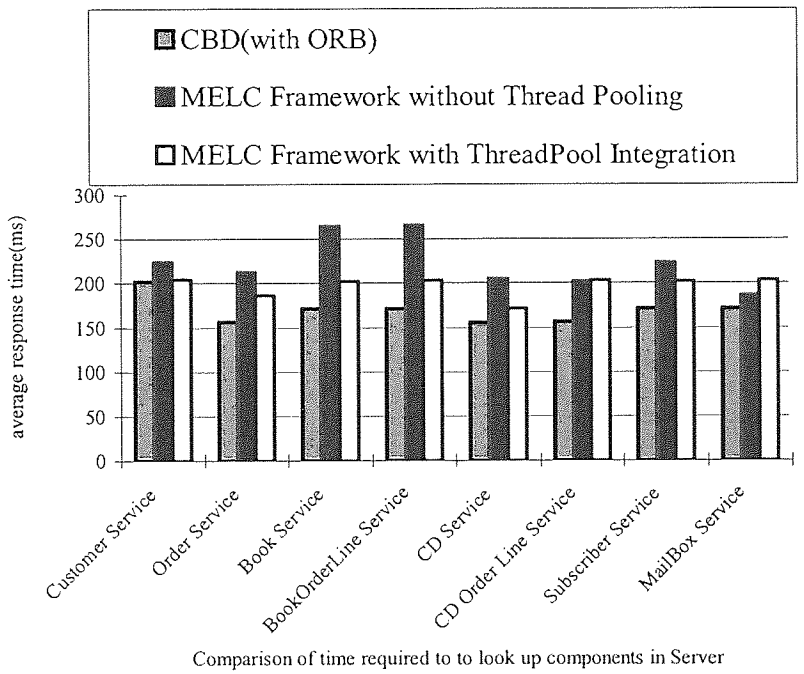

Figure 12.1 ART for *looking up* Remote Business Components

In E-bookshop, we conduct the performance evaluation on the *looking up* operation of the following remote business services from server:

- Customer Service – Customer maintenance
- Order Service – Placing orders
- Book Service – Book Searching and maintenance
- CD Service – CD Searching and maintenance
- Subscriber Service – Customer subscription for sales and
- Mail Box Service – Mail box for mails reception and send.

Figure 12.1 presents the comparison of time required to look up remote business services in server. Table 12.1 summarizes the actual values of the Average Response Time (ART) for the `Look-Up` operation of remote business components via ORB.

Remote Components Look-Up	CBD	MELC - no Thread Pooling	MELC with Thread Pooling Integration
Customer Service	203	225	205
Order Service	157	213	187
Book Service	172	265	203
BookOrderLine Service	172	266	204
CD Service	156	206	172
CD Order Line Service	157	203	204
Subscriber Service	172	224	203
MailBox Service	172	187	204
ART (ms)	170.125	223.625	197.75

Table 12.1 ART for *Looking up* Remote Business Components

Table 12.1 can be further explained as follows. The business services in E-bookshop reside in Object Request Broker (ORB) Registry in a distributed computing environment and are ready for clients to invocate a remote lookup operation. The average response time for looking up an object in a platform using Component-Based Development (without MELC) with ORB [ORB Only] for remote object invocation is 171 milliseconds (ms). In comparison with using MELC, the framework may employ thread pooling component. As the number of threads available in the MELC framework increases, more processing capacity is available to read requests from the sockets and simultaneously process the requests. The Thread Pool component reduces the queue length to access the server process and consequently the average response time for looking up an object dynamically decreases from 223 milliseconds to 197 milliseconds.

In E-bookshop, we also conduct the performance evaluation on the *execution* operation of the following core remote business services in remote server:

- Customer Service – Search customers
- CD Order Service – Search and maintain CD order details
- Book Order Service – Search and maintain book order details

Figure 12.2 presents the Average Response Time (ART) for execution of the remote business components to be executed in between Component-Based Development (without MELC) - we call CBD in short and MELC Framework.

The ART for CBD, MELC without Thread Pooling and MELC with Thread Pooling are 203, 223 and 209 milliseconds (ms) accordingly in Table 12.2. The overhead for adopting the adaptable MELC is relatively very minimal (6 ms) for business components processing.

The analysis of the benchmark results leads to the observation that the level of threading in the container is a key factor for determining the overall contention level in the application. This observation has been empirically validated in both J2EE technologies and CORBA technologies [100]. This means thread pooling must be taken to determine the performance of concurrency in an application. In addition, it shows that the overhead is relatively very minimal (6 ms). The key factors contributed to overhead in the benchmark results are: the kernel architecture in MELC for providing the reflection described in Section 7.3, and the integration of meta components described in Section 10.2, ORB and Thread Pool, at the meta level for providing concurrency in the server. We consider the overhead cost (6 ms) is acceptable in view of having the reflective architecture and concurrency in server.

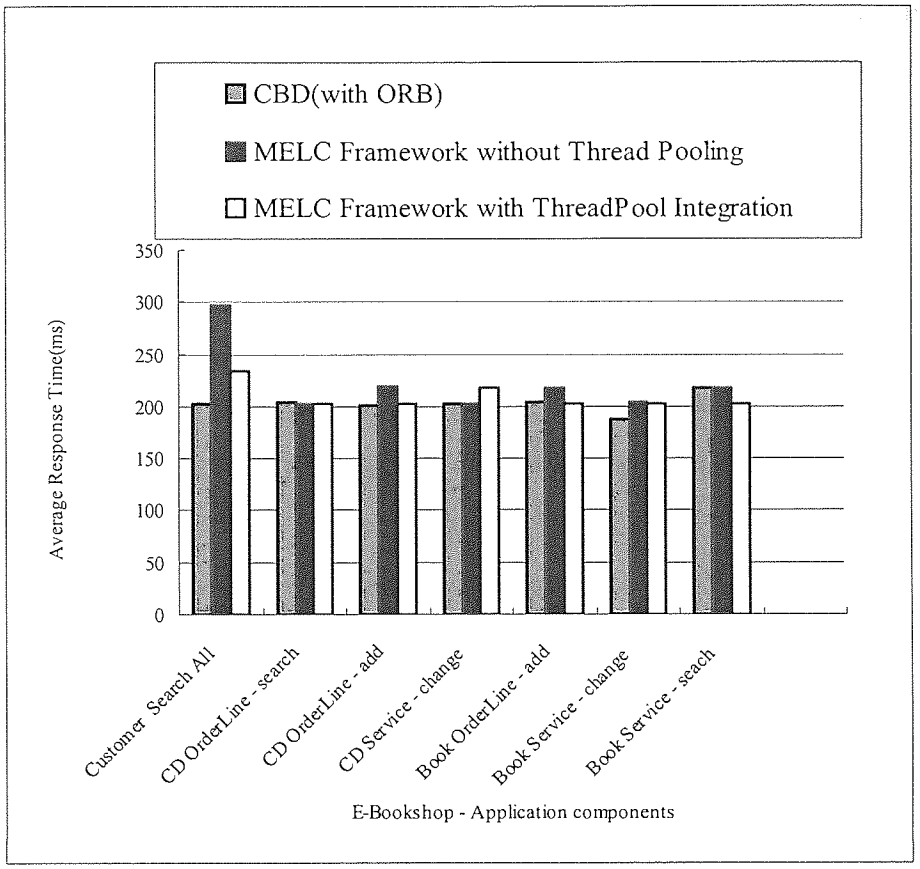

Figure 12.2 Execution Time for remote business components in Server

Execution Business Components	CBD	MELC - no Thread Pooling	MELC with Thread Pooling Integration
Customer Search All	203	297	235
CD OrderLine - search	204	203	203
CD OrderLine - add	201	219	203
CD Service - change	203	203	218
Book OrderLine - add	204	218	203
Book Service - change	188	204	203
Book Service - seach	218	218	203
ART (ms)	203	223.14	209.71

Table 12.2 ART for Execution of the Remote Business Components

Object Request Broker is the primary mechanism for connecting objects between remote address spaces in framework. The summary for the ART of Remote

Business Object *Look-Up* and Remote Business Object *Execution* in the ORB infrastructure is listed in Table 12.3. In the table, we compare the performance of Component-Based Development (without MELC) and the one with the adaptable MELC, and illustrate the average overhead costs of MELC 14% and 3% for *look-up* and *execution* of business objects respectively. The key factor contributed to overhead in the benchmark results is the ORB infrastructure in MELC described in Section 10.4: the ORB Proxy at the base level acts as an ORB Server that receives ORB requests from remote clients, and constructs an ORB operation which is passed to ORB register (meta component) at the meta level. The cost of overhead provides the ORB infrastructure in MELC for transparency for objects distribution and we consider that is reasonable tradeoff.

Remote Business Components	CBD (a)	MELC (b)	Overhead (b) – (a)	Percentage (b – a) / (a)
Look up	171 ms	197 ms	24 ms	14%
Execution	203 ms	209.71 ms	6.71 ms	3%

Table 12.3 MELC overhead costs for Look-up and Execution of Remote Business Components

In addition to remote *business components* for business services in the E-Bookshop application, there are three common *system components* that provide system services in a distributed computing environment and which we have adopted in the implementation of our E-Bookshop application. These components are the Mail Box Service, the Heart Beat Service and the Subscriber/Publisher Service. We have measured the performance of these three key system components in detail and we will now go on to discuss the results.

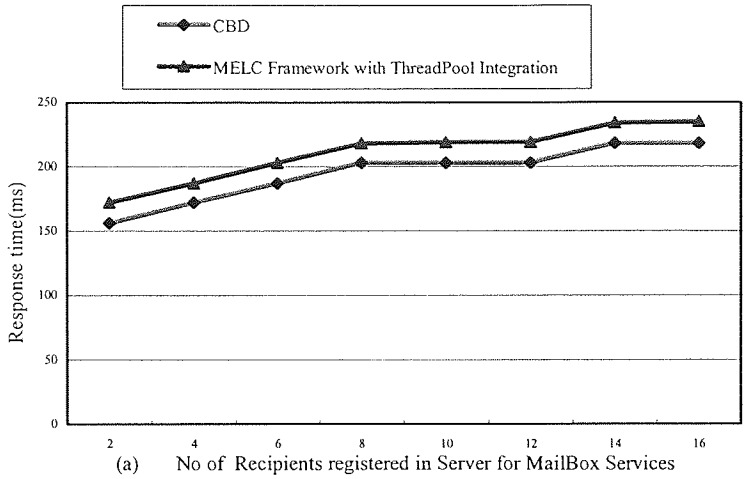

Figure 12.3 (a) Performance Analysis on Mail Box Services

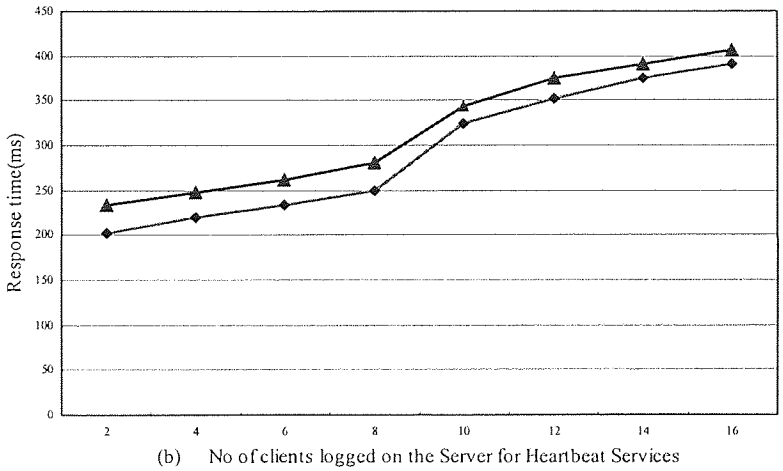

Figure 12.3(b) Performance Analysis on Heart Beat Services

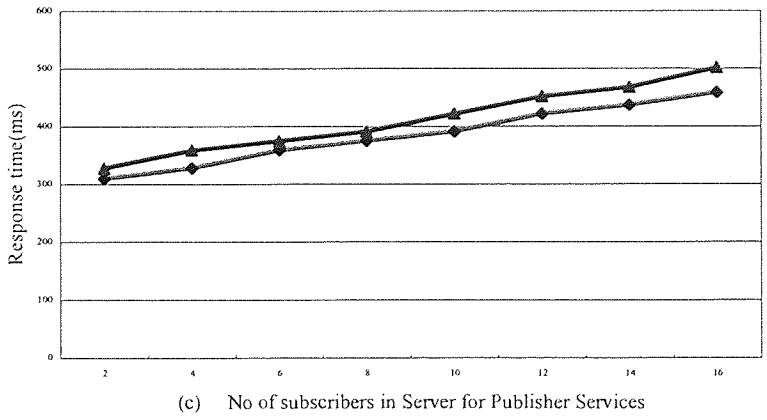

Figure 12.3(c) Performance Analysis on Publisher Services

Mail Box Component: In Figure 12.3(a), as more concurrent recipients are registered in the Mail Box Server, more contention is incurred for the server threads to process the mail requests. Consequently the sender request queue will grow as requests wait for service. From 2 to 16 mail box recipients, the Response Time (RT) for CBD and MELC change from 156 to 218 ms and 172 to 235 ms respectively. An increase in the recipient load on the Mail Box Servers causes decline in the overall throughput of sending and receiving services in the frameworks (CBD and MELC). The difference in Average Response Times (ART) of two frameworks is 16 ms which means the overhead cost of having adaptable MELC for Mail Box Server is approximately 8.2% in terms of response time. The key factors contributed to the overhead in the benchmark results are: the conformation of meta components for adaptability described in Section 10.1 and the separation of concerns of application level and system level described in Section 7.4 and 7.5. In order to work as an adaptable meta component in MELC, the domain specific patterns such as Mail Box must be conformed so that the framework establishes environmental conditions for component instance and regulates the interaction between component instances. In addition, the `start` and `stop` controls at both the base and meta levels provide the separation of concerns between application level and system level in MELC framework. The overhead cost is acceptable (8.2%) in terms of tradeoff for providing separation of concerns in the adaptable layer.

Heart Beat Component: We conduct the test with the same approach as Mail Box Service. In Figure 12.3(b), the application server has from 2 to 16 clients and its heart beat is set at 10-second interval. The clients concurrently receive the heart beat messages sent by their application server. As more concurrent clients log on the server and more tasks run in server and client workstations, more contention is incurred for the clients and the server to process the heart beat messages. With between 2 to 16 clients logged on the server, the Response Times (RT) for CBD

and MELC are recorded to be from 203 to 391 ms and 234 to 406 ms respectively. It has been observed that the overall throughput of heart beat services for the two framework architectures declines by increasing the additional workload and clients to the framework server. The difference in Average Response Times (ART) of two frameworks is 27 ms which means the overhead cost of having adaptable MELC for heart beat service is approximately 9.3% in terms of response time. The factors contributed to the overhead of Heart Beat in the benchmark results are more or less same as Mail Box. The primary difference in the functionalities of Hear Beat in comparison with Mail Box is that the heart beat function is a remote service and performs regularly in a pre-defined interval (say 10-second interval). It has been observed that the time for sending the server heart beat messages to the remote clients triggers the processor of the server into a busy state. We consider the overhead (9.3%) is acceptable in view of the invocation of remote services in an adaptable server.

Subscribe/Publish Component: We conduct the test with the same approach as Mail Box and Heart Beat Services in the framework servers (CBD and MELC). In Figure 12.3(c), as more concurrent subscribers register for the Publisher Services, more contention is incurred for the server. With between 2 to 16 subscribers, the Response Times (RT) for CBD and MELC obviously change from 310 to 459 ms and 328 to 502 ms respectively. The difference in Average Response Times (ART) of two frameworks (CBD and MELC) is 27 ms which means the overhead cost of having adaptable MELC for Subscribe/Publisher Services is approximately 7.1% in terms of response time. The factors contributed to the overhead of Subscribe/Publish in the benchmark results are more or less same as Mail Box and Heart Beat. The association of subscriber and publisher are two dynamic and interactive partnerships. The primary difference is that Subscribe/Publish has the storage for storing the subscribers registered with their interests for publisher services. Messages are delivered to the subscribed recipient objects by transmitting each message to each recipient. Delivery is ensured by repeating the

transmission until successful. As described in Section 11.4, the subscriber details are the Concerned Instances of the old component and the new component in the process of replacement in distributed computing environment at runtime. In addition, the integration of meta components, Publish/Subscriber and Retransmission, at the meta level to ensure delivery of messages to subscribers. We consider the overhead (7.1%) is acceptable in view of having the adaptable server to maintain the concerned instances of the components for version upgrading as system evolves, and supporting the integration of meta components at the meta level.

The differences in Average Response Times (ART) of the system services between CBD and MELC with Thread Pooling are summarized in Table 12.4.

System (meta) Components	CBD (a)	MELC (b)	Overhead (b) – (a)	Percentage (b – a) / (a)
Mail Box	195 ms	211 ms	16 ms	8.2%
Heart Beat	290 ms	317 ms	27 ms	9.3%
Publisher	385 ms	412 ms	27 ms	7.1%

Table 12.4 Performance Analysis of MELC Framework

The overheads for adopting the adaptable MELC for Mail Box Service, Heart Beat Service and Subscriber/Publisher are 8.2%, 9.3% and 7.1% respectively, which averages to be approximately 8.2%. We consider the cost of 8.2% (less than 10%) average response time is acceptable [100] in terms of tradeoff for having a remote adaptable framework – MELC to provide adaptability for system evolution at runtime.

Nonetheless, certain applications may involve large numbers of clients, such as over 100 recipients registering in the server for Mailbox Services in our E-Bookshop application. The overhead for adopting the MELC approach may be different in comparison with the data collected in Figure 12.3. The solution to the problem could be the raw materials for hatching new patterns to replace the existing thread pooling component with kernel manager provided in adaptable MELC framework. Still, other performance improvements are possible and are worth studying, such as improving the architecture design by using model-driven approach and re-factoring the source code.

12.4 Summary

To demonstrate that the MELC framework is implemented efficiently, we have conducted a test and performance analysis for our framework. In the test suite, we compare the performance of two different versions of an E-Bookshop application. One version was implemented using component-based development (CBD) approach which does not use MELC but goes through ORB infrastructure for object distribution – a non adaptable approach, and the other one was implemented using our MELC Framework approach which had reified ORB component at the meta level of meta architecture for object distribution – an adaptable approach.

Our performance tests show the MELC can be implemented with minimal overhead (less than 10%). The study of the results leads to the observation that certain applications may require tighter performance bounds. To this end, we may provide alternative implementations by increasing the workers in thread pool up to the limit of server processor. Still, other performance improvements are possible. We can conclude that, using our adaptable MELC framework, developers can concentrate on creating business logic without being distracted by having to address issues in the domain of distributed computing in system evolution. The adaptable MELC provides a mechanism to facilitate system evolution in the runtime environment.

Chapter 13 Conclusion

In this thesis, we present a new adaptable framework for runtime evolutionary software. While traditional reflective architecture can allow adaptation for selected functional behaviours, we place particular emphasis on the dynamic replacement of system functional behaviours associated with distributed applications, with the aim of allowing for dynamic evolution of running systems.

Our Meta Level Component-Based Framework (MELC) combines a meta architecture with a pattern-oriented framework, resulting in the introduction of an adaptable and configurable layer in the framework which we use to provide a mechanism that facilitates system evolution. This use of a meta architecture with a pattern-oriented framework for distributed computing applications is new and has not previously been explored in research.

Our use of a meta architecture in MELC provides adaptation by way of two key properties: separation of concerns and extensibility. There is separation of concerns in the sense that the base-level objects (i.e. business applications) do not possess any prior knowledge of the identity of the meta-level objects which serve them. Similarly, meta-level objects need not be aware of the base-level objects they serve. Communications between the base level and the meta level in MELC are implemented by the composition connectors in crosscutting. The meta architecture of MELC framework is extensible in the sense that new meta objects can be separately installed, constructed and dynamically integrated by the MELC runtime environment to perform services for requests. This particular architecture represents a new approach to the facilitation of system evolution.

The "reflective" architecture used in other approaches [4, 74, 76, 79] also incorporated a meta architecture but the distributed computing components were hard coded solutions which were interwoven within the architectural elements. Our approach separates the distributed computing elements from the architecture and hence significantly improves the scope for adaptability and extensibility.

Although we have presented our framework using an object-oriented approach, we have illustrated that the architecture may be modularized as components in order to address more specific distributed computing services. For example, an appropriate selection and implementation of distributed computing components may be used to address retransmission issue, object request broker, thread pooling, mailbox, and real-time heart-beat services.

In this thesis, we have described how the architecture of MELC can meet diverse requirements in distributed computing systems, and how the MELC kernel can simply and uniformly create the adaptable components using patterns, and also how those components can be replaced from the framework at runtime in order to provide the adaptability that is most critical in most distributed computing frameworks today.

In addition, MELC provides software decoupling of the system functionality from the business functionality. The system functionality is provided at the meta level and the business functionality at the base level. This makes the system's technological features open-ended for extension, and allows system functions to continually evolve. Our work presented in this thesis has emerged as a promising way to meet challenges of adaptability in the distributed environment currently and in the future.

To demonstrate utility of the framework, we have defined a MELC Programming Model: a programming model that may be used to easily conform the object-oriented classes for a design pattern into a meta component (meta object) that can be incorporated into the meta space of MELC. The computational behaviour of the architecture is defined by the meta space of MELC. In particular, the MELC Programming Model may be used to efficiently implement the operations of reification and de-reification, and of starting and stopping meta objects. The base level manages the individual business application and meta level manages the components at the system level. The operations of a meta object affect the behaviour of all business applications at the base level of the system. The MELC Programming Model provides adaptability of meta objects, which can be replaced at runtime for mission-critical applications.

To demonstrate that the model may be implemented efficiently, we have proposed and implemented a prototype of a typical distributed computing application, E-Bookshop, which forms the backbone of this thesis in explaining MELC from the point of view of the application developer. Although we have based MELC on a simplistic view of resources, we have illustrated that MELC may be enhanced in order to address more specific distributed computing issues. A key concern is the overhead entailed by the component-based adaptability of the MELC framework. Our performance tests show that MELC may be implemented with minimal overhead (less than 10%) which we consider to be acceptable in terms of trade off for providing adaptability at runtime. Still, other performance improvements are possible and are worth studying, such as improving the architecture design with a model-driven approach and applying re-factorization to source codes.

While the specification and implementation of the MELC framework are important issues, it is perhaps more important to derive configuration and management tools which can be used to define the initial configuration and facilitate dynamic coordination and integration of distributed computing components for software applications. Several features are of interest:

- Does the tool verify the existence and property (status) of the meta components at meta space before they are reified by a base object?
- Does the tool transfer the internal structure (concerned instance) of a meta object to a new meta object while processing the attached operation of a meta object for adaptability at runtime?
- Can one meta component be viewed under the base object after the latter has been reified?

As a basis for answering such questions, we have developed a Meta Kernel Configuration Manager. By using aspect-oriented methodology and role-based software engineering, MELC can resolve the meta-level distributed computing components integration. Each component performs a specific distributed computing task and can dynamically interact with other components to have meta object integration to support the specific requests. The design of dynamic integration of meta objects is new and is one of the main features in MELC. In addition, the Meta Kernel Configuration Manager efficiently monitors and administers the component services in the framework. Distributed computing components can be easily built, verified and replaced in the runtime environment for distributed computing applications to access single service or multiple services. The Meta Kernel Configuration Manager provides dynamic services

adaptation and integration in the framework.

We believe that adaptability issues are significant obstacles in the development of distributed computing framework. In particular, current reflective models rely on either cohesive dependency with system objects which obscure too much of the underlying system behaviour, or are limited to anticipated requirements making them unable to perform dynamic adaptation for system evolution. Moreover, such models tend to adapt to anticipated requirements, making it difficult to respond to the unanticipated dynamic evolution of mission critical systems which cannot be taken offline. We believe the MELC framework described in this thesis makes a significant contribution towards adaptable architecture for distributed computing systems. Specifically, the notion of adaptability, transparency, separation of concerns, extensibility and portability preserving pattern-oriented meta-architecture affords the protection of current abstraction boundaries of system behaviour in distributed computing while allowing for graceful integration between components to support system features.

Nonetheless, significant work remains to be done to expand MELC's component base. The types of pattern-based components that we think should be included are: Fault Tolerance, Connection Multiplexing, Heavyweight/Lightweight, and new design patterns, with the latter used to address new issues in the domain of distributed computing. We view such distributed computing components as an evolutionary extension of the MELC described in this work.

Bibliography

[1] E. Gamma, R. Helm, R.Johnson, J. Vissides: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Welsey, Reading, MA, 1995

[2] F. Buschmann, D. Schmidt, M. Stal, H. Rohnert: Pattern-Oriented Software Architecture – A System of Patterns, Volume 1, John Wiley, April 2001

[3] J. A. Broecke, J. O Coplien: Using Design Patterns to Build a Framework for Multimedia Networking, Design patterns in Communication Software, SIGS, Cambridge University Press 2002

[4] J. Suzuki, Y. Yamamoto: OpenWebServer: an Adaptive Web Server using Software Patterns. IEEE Communications Magazine, April 1999.

[5] L. Fuentes, J.M. Troya: A JAVA Framework for Web-Based Multimedia and Collaborative Applications. Internet Computing, March-April 1999

[6] S.M. Yacoub, H. H. Ammar: Toward Patter-Oriented Frameworks. Journal of Object-Oriented Programming, January 2000

[7] G. Booch, J. Rumbaugh, I. Jacobson: The Unified Modeling Language User Guide. Addison-Wesley, 1999.

[8] M. Fowler: UML Distilled. 3rd Edition. Reading, Mass., Addison-Wesley, 2004.

[9] G. Booch, J. Rumbaugh, I. Jacobson: The Unified Modeling Language Reference Manual. Addison-Wesley, 1999.

[10] H.E. Eriksson, M. Penker: UML Toolkit. New York, John Wiley & Sons, 1998.

[11] G. Booch, J. Rumbaugh, I. Jacobson: The Unified Modeling Development Process. Addison-Wesley, 1999.

[12] R. Pooley, P. Stevens: Using UML. Reading, Mass., Addison-Wesley, 1999.

[13] X. Jia: Object-Oriented Software Development Using Java - Principles, Patterns, and Frameworks, 2nd Edition, Addison-Wesley, 2003.

[14] M. Grand: Java Enterprise Design Patterns, John Wiley, 2002

[15] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, S. Angel: A Pattern Language – Towns, Building, Construction. Oxford University Press, New York, 1977.

[16] W. Pree: Design Patterns for Object-Oriented Software Development, Reading,

Mass., Addison-Wesley, 1995.

[17] Alpert, Sherman, K. Brown, B. Woolf: The Design Patterns Smalltalk Companion. Reading, Mass.: Addison-Wesley, 1998.

[18] Buschmann, Frank, R. Meunier, et al: Pattern Oriented Software Architecture: A System of Patterns. New York, John Wiley, 1996.

[19] O.J. Coplien, D. Schmidt: Pattern Languages of Program Design. Reading, Mass., Addison-Wesley, 1995.

[20] M.E. Fayad, D.C. Schmidt: Object-Oriented Application Frameworks, Communicatons of the ACM, October 1997.

[21] R.E. Johnson, B. Foote: Designing Reusable Classes. Journal of Object-Oriented Programming, January-February, 1988.

[22] R.E. Johnson, B. Foote: Designing Reusable Classes, Journal of Object-Oriented Programming, June-July 1988.

[23] H. A. Schmid, F. Mueller: Patterns for Extending Back-Box Frameworks, Journal of Object-Oriented Programming, June 1998.

[24] D. D'Souza: Interface Specification, Refinement, and Design with UML/Catalysis, Journal of Object-Oriented Programming, June 1998.

[25] H. A. Schmid: Creating Applications from Components: Manufacturing Framework Design, IEEE Software, Nov 1996

[26] G.F. Rogers: Framework-Based Software Development in C++, Prentice Hall, Englewood Cliffs, NJ, 1997.

[27] M.E. Fayad, D.C. Schmidt,Ralph E. Johnson: Building Application Framework, John Wiley, 1999.

[28] M. Boger: Java in Distributed Systems, John Wiley, 2001

[29] Hassen Gomaa: Designing Concurrent, Distributed, and Real-Time Applications with UML, Addison Wesley, 2000.

[30] M.L. Liu: Distributed Computing Principles and Applications, Pearson Addison-Wesley, 2003

[31] R.Orfali, D. Harkey, J.Edwards: Client/Server Survival Guide, 3rd Ed., New York, John Wiley & Sons, 1999.

[32] J. Zukowski: Mastering Java 2, SYBEX Inc, 1998.

[33] M. Wutka: Using Java 2 Enterprise Edition, Special Edition, QUE, 2001.

[34] D. Box: Essential COM. Reading, Mass., Addison-Wesley, 1998.

[35] Y.P. Shan, R.H. Earle: Enterprise Computing with Objects. Reading, Mass., Addison-Wesley, 1998

[36] Sun Microsystems, Enterprise JavaBeans Specification Version 3.0, http://java.sun.com/products/ejb/, Last Updated: May 2005

[37] Microsoft, http://www.microsoft.com/com/default.mspx, COM Home Page, Last Updated: 2005

[38] Jordi Alvarez Canal: Parametric Aspects: A Proposal, ECCOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution, 2004.

[39] Nikos Parlavantzas, Geoff Coulson, Mike Clarke, and Gordon Blair: Towards a Reflective Component Based Middleware Architecture, ECCOOP Workshop on Reflection and Metalevel Architectures, 2000

[40] Object Management Group, CORBA Components V 3 Full Specification, http://www.omg.org/technology/documents/formal/components.htm, Last updated: May 26, 2005

[41] O. Ciupke, R. Schmidt: Components as Context-independent Units of Software. In Processing of ECOOP, 1996.

[42] A.H. Eden, J. Gil, A. Yedudai: A Formal Language for Desing Patterns. In Processing of the 3rd Annual Conference on the Pattern Languages of Programs, 1996.

[43] C. Alexander: The timeless way of Building, Oxford University Press, New York, 1979.

[44] Magnus Larsson: The Different Aspects of Component Based Systems, The Component-based Software Engineering, State of the Art Report, Vasteras, Sweden, 2000

[45] Z. Kiziltan, T. Jonsson, B. Hnich: On the definition of Concepts in Component Based Software Development, The Component-based Software Engineering, State of the Art Report, Vasteras, Sweden, 2000

[46] M. Blom, E.J. Nordby: Semantic Integrity in Component Based Development,

The Component-based Software Engineering, State of the Art Report, Vasteras, Sweden, 2000

[47] Martin P. Robillard: Separation of Concerns and Software Components, Report of The Component-based Software Engineering – State of the Art, pages 54-68, Malardalen University, Department of Computer Engineering, Vasteras, Sweden, March 2000

[48] William Harrison and Harold Ossher: Subject-oriented programming (a critique of pure objects). In Proceedings of the conference on Object-Oriented Programming Systems, Languages, and Applications (OOSPLA '93), pages 411-428. ACM SIGPLAN, 1993.

[49] Gregor Kiczales, John Lamping, Anurag Mendhekar et al: Aspect-oriented programming. Proceedings of the European Conference on Object-Oriented Programming (ECOOP), volume 1241 of Lecture Notes in Computer Science, pages 220-242, Springer-Verlag, June 1997.

[50] Mehmet Aksit and Bedir Tekinerdogan: Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters. ECOOP'98 Workshop Reader, volume 1543 of Lecture Notes in Computer Science, Springer-Verlag, July 1998

[51] Harold Ossher, William Harrison, Frank Budinsky, and Ian Simmonds: Subject-oriented programming: Supporting decentralized development of objects. In proceedins fo the 7th IBM Conference on Object-oriented Technology, IBM, July 1994.

[52] Ivar Jacobson and Pan-Wei Ng: Aspect-Oriented Software Development with Use Cases, Addison-Wesley, 2005

[53] Kiczales, G., Hilsdale, E. Hugunin, J., Kersen, M. Palm, J., Giswold. W.: Getting Started with AspectJ. Communications of the ACM 59-65, 2001.

[54] AspectJ Team: AspectJ Programming Guide Http://www.eclipse.org/Aspectj, Last Updated 2005.

[55] Mehmet Aksit and Bedir Tekinerdogan, TRESE Project, University of Twente,Centre for Telematics and Information Technology, 2001

[56] Mik A. Kersten and Gail C. Murphy. Atlas: A case study in building a web-based learning environment using aspect-oriented programming. In Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, 1999.

[57] M. P. Robillard: Separation of Concerns and Software Components, The

Component-based Software Engineering, State of the Art Report, Vasteras, Sweden, 2000

[58] Carnegie Melon - Software Engineering Institute, CBSD Integration, http://www.sei.cmu.edu/str/descriptions/cbsd_body.html

[59] S.M. Yacoub, H. H. Ammar: Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems, Addison Wesley Professional, August 2003.

[60] K. Beck, R. Johnson: Patterns Generate Architectures, ECOOP '94

[61] R.E. Johnson: Frameworks = Components + Patterns, Communications of the ACM, October 1997.

[62] X. Castellani, S.X.Liao: Development Process for the Creation and Reuse of Object-Oriented Generic Applications and Components, Journal of Object-Oriented Programming, June 1998.

[63] G. Odenthal, K. Quibeldey-Cirkel: Using Patterns for Design and Documentation, Proceedings of the 11th European Conference of Object-Oriented Programming, 1997

[64] H. A. Schmid: Creating the Architecture of a Manufacturing Framework by Design Patterns, Proceedings of Object-Oriented Programming Systems, Lanuages and Applications, OOPSLA'95, 1995

[65] G. Agha, D.C. Sturman: A Methodology for Adapting to Patterns of Faults, Foundation of Ultradependability, Vol 1, Kluwer Adademic 1994.

[66] S. Duncasse and T.Richner. Executable Connectors: Towards Reusable Design Elements. In Proceedings of the European Software Engineering Conference (ESEC'97), volume 1310 of Lecture Notes in Computer Science, pages 484-500, September 1997, Springer-Verlag.

[67] C.P. Lunau: A Reflective Architecture for Process Control Applications. Proceedings of the 11th European Conference on Object Oriented Programming (ECOOP'97), volume 1241 of Lecture Notes in Computer Science, Berlin Germany, June 1997. Springer-Verlag.

[68] R.J. Walker and G.C. Murphy. Dynamic Contextual Reflection: A Mechanism for Software Evolution and Reuse. In Proceedings of the OOPSLA Workshop on Object Oriented Reflection and Software Engineering (OORaSE'99), Denver, November 1999.

[69] N. Amano and T. Watanabe. An Approach for Constructing Dynamically Adaptable Component-based Software Systems using LEAD++. In Proceedings

of the OOPSLA Workshop on Object Oriented Reflection and Software Engineering (OORaSE'99), Denver, CO, November 1999.

[70] R. de Lemos and E. Tramontana: A Reflective Implementation of Software Architectures for Adaptive Systems. In Proceedings of the Second Nordic Workshop on Software Architectures (NOSA'99), Ronneby, Sweden, 1999.

[71] Emiliano Tramontana: Reflective Architecture for Changing Objects, Proceedings of the Conference on Object Oriented Programming (ECOOP'00), volume 1964 of Lecture Notes in Computer Science, June 2000. Springer-Verlag.

[72] U. Assmann: Invasive Software Composition, Springer-Verlag, 2003

[73] U. Assmann et al.: Automated Component-Based Software Engineering, Journal of Systems and Software 74 1-3, Elsevier, 2005

[74] Barry Redmond, Vinny Cahill: Iguana/J:Towards a Dynamic and Efficient Reflective Architecture for Java, ECCOOP Workshop on Reflection and Metalevel Architectures, 2000

[75] Brendan Gowing, Vinny Cahill: Meta-Object Protocols for C++: The Iguana Approach, Reflection'96 , 1996

[76] Ayla Dantas, Paulo Borba, Joseph Yoder and Ralph Johnson: Using Aspects to Make Adaptable Object-Models Adaptable, ECCOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution, 2004.

[77] Yoder, J.W., Johnson, R.: The Adaptive Object-Model Architectural Style. In Working IEEE/IFIP Conference on Software Architecture 2002(WICSA), Montreal, Quebec, Canada (2002)

[78] Yoder, J.W. Balaguer, F., Johnson, R.: Architecture and Design of Adaptive Object-Models. ACM SIGPLAN Notices 36 (2001) 50-60

[79] Ruzanna Chitchyan and Ian Sommerville: AOP and Reflection for Dynamic Hyperslices, ECCOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution, 2004.

[80] H. Ossher and P. Tarr: Multi-Dimensional Separation of Concerns using Hyperspaces and Hyper/J User and Installation Manual, IBM Research, 2000

[81] Walter Cassola, Ahmed Ghoneim, Gunter Saake: RAMSES: a Reflective Middleware for Software Evolution, ECCOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution, 2004.

[82] Yoshiki Sato and Shigeru Chiba: Negligent Class Loaders for Software

Evolution, , ECCOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution, 2004.

[83] JBoss Open Source, UCL Class Specification, http://wiki.jboss.org web link, Last Updated: July 2005

[84] Nelly Bencomo, Gordon Blair, Geoff Coulson and Thais Batista: Towards a Meta-Modelling Approach to Configurable Middleware, RAM-SE'05 2nd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution, Glasgow, Scotland, 25th of July 2005.

[85] Andy S.Y. Lai, A.J. Beaumont: Meta-based Distributed Computing Framework, Lecture Notes of Computer Science, Parallel and Distributed Processing and Applications, ISPA2004, LNCS 3358, pp85-90, Springer-Verlag, December 2004.

[86] Andy S.Y. Lai, A.J. Beaumont: A Metalevel Component-Based Framework for Distributed Computing Applications, Fourth Annual ACIS International Conference on Compute and Information Science (ICIS'05), IEEE, Journal of Computer Society, p268-273, April 2005.

[87] Ian Welch, Robert Stroud: From Dalang to Kava – the Evolution of a Reflective Java Extension, Second International Conference, Reflection '99, LNCS 1616, pp2-21, Springer-Verlag, July 1999.

[88] Andy S.Y. Lai, A.J. Beaumont: Development of Enterprise WAP E-Banking, Conference of International Telecommunications Society (ITS2002), 14th Biennial Conference, Seoul, Korea, August 2002.

[89] Andy S.Y. Lai: An Object-Oriented Approach for Building Distance Learning System on Internet, The 6th Asia Pacific Regional Conference of International Telecommunications Society, Hong Kong, July 2001

[90] Andy S.Y. Lai: Building Web Time-Series Forecasting Systems on Stocks with UML, Fifth ICSA International Conference, Hong Kong, August 2001.

[91] K. Brown, B Whitenack: Crossing Chasms: A Pattern Language for Object-RDMS Integration 'The Static Patterns', 1998.

[92] F. Buschmann et al.: A System of Patterns, John Wiley, 1996

[93] W. Doeringer et al.: A Survey of Light-Weight Transport Protocols for High-Speed Networks, IEEE Transaction on Communication, Volume 3, 1990.

[94] P. Sommerlad, M. Ruedi: Do-it-yourself Reflection Patterns, EuroPLoP, 1998.

[95] Doug Lea: PooledExecutor – http://gee.cs/oswego.edu/dl/classes/EDU/oswego/cs/util/concurrent/

[96] M. Grand: Patterns in JAVA, Volume 2, John Wiley, 1999

[97] T. Larsson, M. Sandberg: Building Flexible Components Based on Design Patterns, The Component-based Software Engineering, State of the Art Report, Vasteras, Sweden, 2000

[98] D'Souza D. F and Wills A.C.: Objects, Components and Framework with UML. The Catalysis Approach, Addison-Wesley, 1999

[99] Paul Hyde: Java Thread Programming – The Authoritative Solution, SAMS, 1999

[100] S. Chen, Y. Liu, I. Gorton, A Liu: Performance Prediction of Component-Based Applications, Elsevier, The Journal of Systems and Software 74(2005) 35-43

[101] T. Gu, H.K. Pung, D.Q. Shang: A service-oriented middleware for building context-aware services, Journal of Network and Computer Applications 20 1-18, Elsevier, 2005

[102] David S. Frandel: Model Driven Architecture – Applying MDA to Enterprise Computing, OMG Press, John Wiley, 2003

[103] Peter Ebraert and Tom Tourwe: A Reflective Approach to Dynamic Software Evolution, , ECCOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution, 2004.

[104] Graham Hamilton, Ricj Cattell and Maydene Fisher: JDBC Database Access with Data, http://java.sun.com/docs/books/tutorial/jdbc/TOC.html, web link, last Updated: 2007 Sun Microsystems, Inc

[105] D. Hughes, P. Greenwood, G.S. Blair, G. Coulson, P. Grace, F. Pappenberger, P. Smith, K. Beven: An Experiment with Reflective Middleware to Support Grid-based Flood Monitoring, Concurrency and Computation: Practice and Experience, 2007.

[106] Kevin Lee, Geoffrey Coulson: Supporting Runtime Reconfiguration on Network Processors", Journal of Interconnection Networks, Vol 7, No 4, , Special Issue on Information Networking and P2P Systems, World Scientific Publishing Co., pp 475-492, 2006.

[107] P. Costa, G. Coulson, C. Mascolo, L. Mottola, G.P. Picco, S. Zachariadis: A Reconfigurable Component-based Middleware for Networked Embedded Systems", International Journal of Wireless Information Networks, Vol 14, No 2, pp 149-162, June 2007.

[108] Nikos Parlavantzas, Geoff Coulson: Designing and Constructing Modifiable Middleware using Component Frameworks", IET Software, Vol 1, No 4, pp 113-126, Aug 2007.

[109] Petros Pissias, G. Coulson, A. Joolia: Supporting Dynamic Reconfiguration in Multithreaded Component-based Systems, IET Software, 2008.

[110] Gordon Blair, Paul Grace, Francois Taiani, Ackbar Joolia, Kevin Lee, Jo Ueyama, T. Sivaharan: A Generic Component Model for Building Systems Software, ACM Transactions on Computer Systems, Vol. 26, No. 1, Feb 2008.

[111] Frank Buschmann, Kevlin Henney, Douglas C. Schmidt: Pattern Oriented Software Architecture: A Pattern Language for Distributed Computing, New York, John Wiley, 2007.

Appendix A - MELC Configuration and Management

A.1 MELC Configuration and Management Utility

This section shows the utility we have developed for configurating and managing meta space. The tool allows the administrator to instantiate meta objects, registers them into the meta object repository, which, once created, also allows run time replacement of meta objects. As for the base level, the configuration and management utility lets the administrator create base objects, attach them to the meta space, and to reify or de-reify meta objects existing in the meta object repository. That interface controls base objects with resume and stop operations. Figure A.4 shows the GUI of the configuration and management utility.

The utility allows the administrator to select and load an existing configuration file, and thereby to re-construct the meta space configuration to the status specified in that configuration file. The utility also allows the current status of MELC to be saved, thereby creating a new configuration file. We have tested the configuration of the minimum set of functionalities for execution environments like the E-Bookshop Thread Pooling Http Server. In such an environment, the E-Bookshop Server (a base object) is small and has limited distributed computing components. The MELC allows the web server to be configured with a minimal set of functionalities, which includes processing HTTP requests without Thread Pool components. During the testing, the utility properly demonstrates administration functions in the kernel.

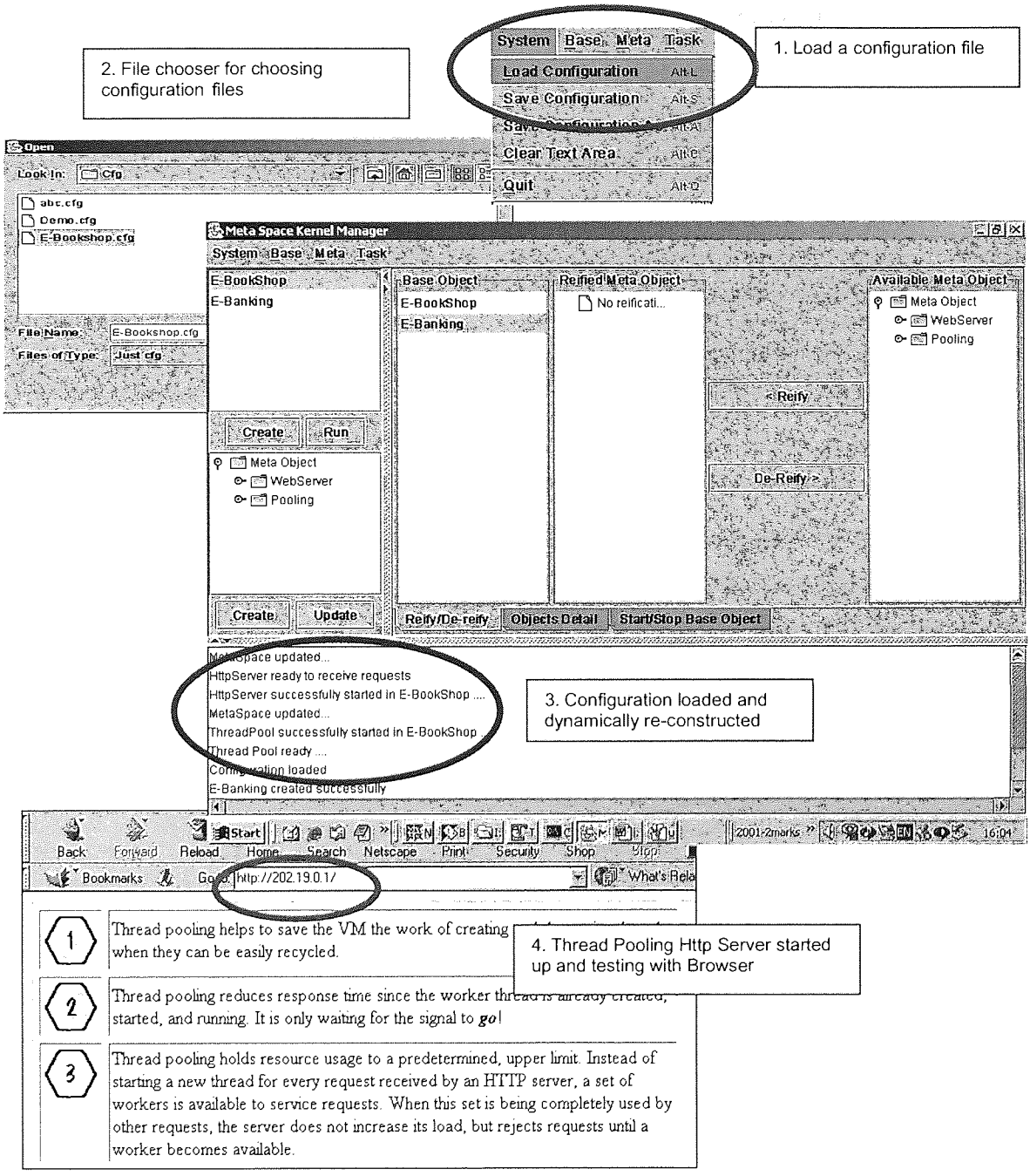

Figure A.1 MELC Kernel Manager - Utility for Configuration Management

Figure A.1 illustrates how a MELC administrator can easily reconstruct the meta space configuration to the status saved in a configuration file. First of all, the framework administrator selects the submenu Load Configuration under System Menu Bar of the Kernel Manager, and the configuration file chooser will pop up and show all existing configuration files in the system for the administrator to select. After the administrator has selected one of them, MELC Kernel Manager dynamically loads the designated file and re-constructs the meta space configuration to the status specified in that configuration file. In our case, an E-Banking Thread Pooling Http Server has been started up in MELC and is ready for handling requests.

Our meta-based framework supports multiple applications. The framework can be applied to different distributed object-requestor applications such as E-Bookshop, E-Banking. The separation of system functionality and business functionality means that business functions of an application are published to base objects (or base servers) at the base level.

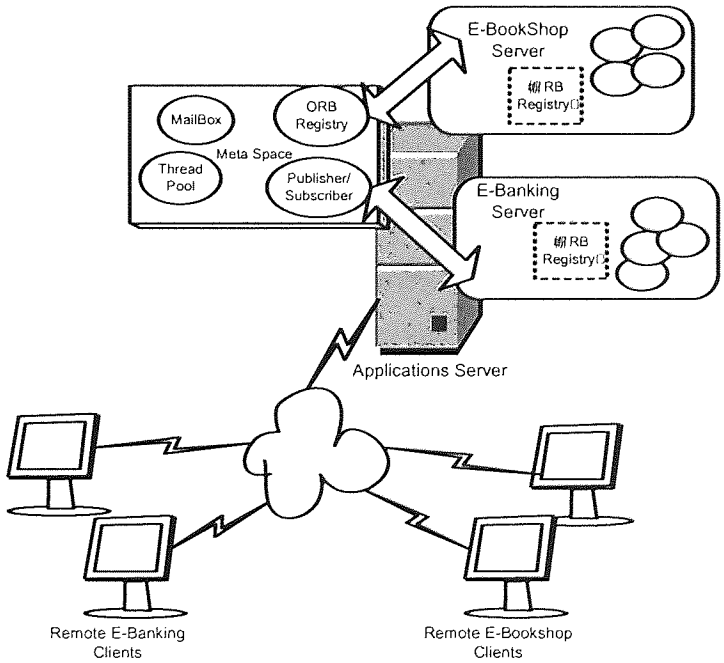

Figure A.2 Meta-based framework supports Multi-Applications

Figure A.2 shows a typical example of the Application Objects of E-Bookshop. The remote objects published by a remote workstation via the network to a base object (which we have called E-Bookshop Server) might include; E-bookshop Customer Entity, E-Bookshop Mail Services, E-Bookshop Ordering and E-Bookshop Book/CD Administration. For an E-Banking Application, those objects will be different, possibly including E-Banking Payment, E-Banking-Saving and would be sent to the base object called E-Banking Server. Meta space provides system functionality to such applications.

Appendix B – Java Classes and their Methods in MELC

B.1 Java Classes and Methods in Kernel

Kernel Manager Functions	**Method**
add base object	`Modules.addBaseObject(..)`
add meta object	`Modules.addMetaObject(..)`
replace meta object	`Modules.changeMetaObject(..)`
reify meta object by base object	`Modules.reifyByBaseObject(..)`
dereify meta object by base object	`Modules.deReifyByBaseObject(..)`
start base object running	`Modules.startBaseObject(..)`
stop base object running	`Modules.stopBaseObject(..)`
start meta object running	`Modules.startMetaObject(..)`
stop meta object running	`Modules.stopMetaObject(..)`
Meta Space Management	**Method**
find base object	`MetaSpace.findBaseObject(..)`
find meta object	`MetaSpace.findMetaObject(..)`
attach meta object	`MetaSpace.attachObject(..)`
detach meta object	`MetaSpace.detachObject(..)`
notify base object	`MetaSpace.notifyBaseObject(..)`
check meta object started	`MetaSpace.isMetaObjectStarted(..)`
check meta object reified in base objects	`MetaSpace.isMetaObjectInAnyBaseObject(..)`
get role partnership	`MetaSpace.getRolePartner(..)`
make role object for meta objects integration	`MetaSpace.makeRoleObject(..)`
retrieve handoff target box	`MetaSpace.getTargetObject(..)`

Table B.1 Kernel Manager Functions and Meta Space Management

B.2 Java Classes and Methods at Meta level and Base Level

Meta Object	**Method**
start meta object	`metaObject.startRequest()`
stop meta object	`metaObject.stopRequest()`
make meta proxy	`metaObject.makeMetaObjectProxy`
Meta Object Proxy	**Method**
start meta object at baselevel	`metaObjectProxy.startRequest()`
stop meta object at baselevel	`metaObjectProxy.stopRequest()`
release resource	`metaObjectProxy.releaseResource()`
set meta type	`metaObjectProxy.setMetaType(..)`
set meta name	`metaObjectProxy.setMetaName(..)`
set base object	`metaObjectProxy.setBaseObject(..)`
Base Object	**Method**
accept meta object	`baseObject.acceptMetaObject(..)`
withdraw meta object	`baseObject.refuseMetaObject(..)`
check meta object reified	`baseObject.isMetaObjectExisted(..)`
retrieve meta object proxy	`baseObject.findMetaObjectProxy(..)`
synchronize actions	`baseObject.updateAndRun(..)`
ORB Handler	**Method**
bind orb object	`Registy.bind(..)`
unbind orb object	`Registy.unbind(..)`
get skeleton table	`ORBRegister.getSkeletonHashtable(..)`
set orb object to a skeleton	`skeletonHashTable.setSharable(..)`

Table B.2 Essential Methods - Meta Object, Base Object and ORB Handler

Appendix C – Java Coding in MELC

C. 11 Process of reification - Base Object Reification of a Meta Object

The following coding shows how the Kernel Manager reifies a meta object for the base object at the base level:

```
//validate if meta components in base object
MetaObject mobj = MetaSpace.findMetaObject(mobjType, mobjName);
BaseObject bobj = MetaSpace.findBaseObject(bobjName);

if ( bobj.isMetaObjectExisted(mobjType, mobj.toString()) ) {
  // "Error: Meta type found in baseObject " + bobj.toString()
}

//assign cache instances in its meta proxy object resided at base level
MetaObjectProxy bpobj = mobj.getProxyObject();
bpobj.setMetaType(mobjType);
bpobj.setMetaName(mobjName);
bpobj.setBaseObject(bobj);

//bind the meta object in case it was unbinded
Registry.getRegistry().bind(mobj.toString(), (Sharable)mobj);

//notify the base object to accept it
bobj.acceptMetaObject(mobjType, mobj.toString(), bpobj);
```

C. 12 Integration between HTTPServer and ThreadPool

In this way, handling tasks depend on the number of workers in `ThreadPool`. The integration between `HTTPServer` and `ThreadPool` has been implemented with Java. The pseudo codes and sources are presented as following.

The *HTTP Server* manages the requests in the following sequence:

1. Http Server accepts socket requests.
2. Check with the base object to see if Thread Pool having been reified by base object.
3. If Thread Pool being reified, Http Server assigns Http Worker to the Target Object (Connector) of Thread Pool in Meta Space, where Target Object is the compositional connector.

The *HTTP Server* has the following program fragment for components integration in *MELC*:

```
while (noStopRequested) {

 try {

    //Http Server accepts socket and creates worker for it
    Socket s = ss.accept();
   HttpWork worker = new HttpWork(docRoot, s);

    //Check Base object (bobj) to see if Thread Pool reified
    if (bobj.findTargetObject("ThreadPool"))  {

        // Now, ThreadPool integrates with HTTPServer …
        // Get the ThreadPool connector (target object)
        TargetObject targetObject =
        metaspace.getTargetObject("ThreadPool");

        //Assigns HttpWorker to ThreadPool connector
        targetObject.objectType = "HttpWork";
        try{
              targetObject.targets.add(worker);
        } catch (InterruptedException ix) {}

    }
    else {
     //HttpWorker processes requests itself
```

```
        worker.runWork();
      }

    }
    catch (IOException iox) {}
  }
```

The *Thread Pooling* integrates with *Http Server* in the following sequence:

1. Thread Pool associates with its Target Object in meta space.
2. Thread Pool removes the requests from Target Object if there is one.
3. Thread Pool assigns a Thread Pool worker to execute the request on the behalf of the *Http Server*.

The *Thread Pooling* has the following program fragment for components integration in MELC:

```
//the method following is embedded in a Thread
private void runWork() {

    //Firstly, Thread Pool retrieves the connector object
    targetObject = metaspace.getTargetObject("ThreadPool");

    // Thread Pool is ready …
    while (noStopRequested) {
      try {
        // Removes and executes the requests from connector object
        RunObject o = RunObject) targetObject.targets.remove();
        execute(o);
      }
      catch (Exception iox) {}
    }

}

//the method is used to execute the requests with a thread pool worker
public void execute(RunObject target) throws InterruptedException {

    // block until a thread pool worker is available
    ThreadPoolWorker worker = ThreadPoolWorker)idleWorkers.remove();

    // let the workers handle the requests
    worker.process(target);
}
```

C. 13 The Role-Based Approach for Components Integration

The following Java coding shows the role object handling the integration between meta object and its partner components in meta space at runtime:

```
//Http Server accepts socket and creates worker for it
HttpWork worker = new HttpWork(docRoot, s);

/*
Let the Role Manager (MetaSpace) to make a role object
where Base Object (bobj) in our example will be E-BookShop.
It has all its reified meta objects.
*/

RoleObject roleObject = MetaSpace.makeRoleObject(bobj,this);

//Check if there is any partners for HttpServer
if (roleObject.isIntegrated()){
    //yes. Then let the role object to cooperate with them
    roleObject.cooperate(worker);
}
else {
    //no. execute by itself
    worker.runWork();
}
```

C. 14 The Remote Method Invocation in Object Request Brokers

The following is the Java program in `ORB Proxy` for method invocation in ORB:

```
protected void runWork() {

   try{

       //start a new server
       ServerSocket ss = new ServerSocket(portNo);

       while (noStopRequested) {

           //accept a networking socket
           Socket s = ss.accept();

           //read the socket to extract an remote request operation
           ObjectInputStream clientIn  =
              new ObjectInputStream(s.getInputStream() );
           Object object = clientIn.readObject();
           operation = (Operation)object;

           //check if the request operation is valid
           if (isValidRequest(operation)) {

             //valid. Retrieve connector TargetObject of ORB
             TargetObject targetObject =
               MetaSpace.getTargetObject(metaName);
            BTargetObject bTargetObject = new BTargetObject();

             //prepare details (socket, operation, base object)
            bTargetObject.socket = s;
            bTargetObject.objectType = "Operation";
            bTargetObject.bobj = bobj;

            //pass operation to the connector of ORBRegister
            try{
                 bTargetObject.targets.add(operation);
                 targetObject.targets.add(bTargetObject);
            } catch (InterruptedException ix) {}

           }
       }
   }catch (Exception ix) {}

}//end of runWork
```

In the Java source code, the invocation operation, `Operation` object, is embedded in the connector, `TargetObject`, and is transported to `ORBRegister` at meta level and allows the `ORBDispatcher` to dispatch to the meta objects, `Mailbox` or `Publisher/Subscriber`, for the required services processing. Note that the `TargetObject` in the source code above is a *composition connector* between base level and meta level, which has been discussed in section 9.4.1. The design and implementation of meta component Object Request Broker in MELC adds the *transparency* of object distribution to our framework.

C.15 Dispatching Remote Requests in Object Request Brokers

The Java source codes for the meta object `ORBRegister` to dispatch the remote requests at meta level have been extracted as following:

```
Protected void runWork() {

    while (noStopRequested) {

        try {

          //Meta Space has ORB TargetObject
          TargetObject  targetObject =
            MetaSpace.getTargetObject(this.toString());

          //Remove whenever there is Target Object available
          BTargetObject  bTargetObject = (BTargetObject )
            targetObject.targets.remove();

          //Retrieve base object (E-Bookshop)
          BaseObject bobj = bTargetObject.bobj;

          //Retrieve socket for returning result
          Socket clientSocket = bTargetObject.socket;

          //Retrieve the operation - remote message
          Operation clientOperation = (Operation)
          bTargetObject.targets.remove();

          //Pass the Operation to ORBDispatcher for dispatching
          //to the proper meta objects at the Meta Level.
          ORBDispatcher worker =   new ORBDispatcher
             (registryHost, registryPort, skeletonHashtable,
              clientSocket, clientOperation);

          worker.runWork();

        } catch (Exception ix) {}

    }
}
```

Note the object of the invocation operation of ORB is embedded in the connector (Target Object). Whenever ORB Meta Proxy puts the `Operation` object to ORB target object in Meta Repository, the ORB Meta Object immediately removes its target object from Meta Repository in the meta space and processes the embedded `Operation`, and ORB meta object dispatches it to the relevant meta object (such as Mailbox or Publisher/Subscriber) for processing.

In the meta space, `ORBRegister` locates and retrieves its connector `TargetObject` from the meta repository in meta space. Inside the `TargetObject`, the `ORBRegister` can extract the information of the remote request: the base object (ie. `E-Bookshop`), the request's socket, and most important the `Operation` (remote request) for processing. After that, the `ORBRegister` assigns its work to `ORBDispatcher` to process the `Operation` obtained from the `TargetObject`, where `ORBDispatcher` dispatches the `Operation` to one of the meta (system) components like *Mailbox* or *Publisher/Subscriber* in `E-Bookshop` for processing.

C.16 Replacement of Meta Objects at Runtime

The following Java source codes show the implementation of replacing meta objects at runtime:

```
//construct new meta object
MetaObjectImpl mobj =
    (MetaObjectImpl) Class.forName(newClassName).newInstance();

//find old meta object in meta space
MetaObject OldMO = MetaSpace.findMetaObject(classType, className);

//check if same Meta Type and Meta ID
if ( !mobj.toString().equals(OldMO.toString()) )  {
    return;}

//***** Old Meta Object *****

// Get concerned instances - skeleton hashtable and port #
TargetObject targetObject =
        MetaSpace.getTargetObject(OldMO.toString());
HashMap skeletonHashtable = targetObject.skeletonHashtable;
int lastPortNoUsed = targetObject.lastPortNoUsed;

//detach old metaobject in Meta Repository
MetaSpace.detachObject(className, classType) )

//***** New Meta Object *****

//attach New meta object in Meta Repository
MetaSpace.attachObject(mobj, classType) )

// Set concerned instances - skeleton hashtable and port #
targetObject = MetaSpace.getTargetObject(mobj.toString());
targetObject.lastPortNoUsed = lastPortNoUsed;
targetObject.skeletonHashtable = skeletonHashtable;

//***** ORB Register *******
//rebind the meta object in ORB Registry
Registry.getRegistry().bind(mobj.toString(), (Sharable)mobj);

//retrieve ORBRegister
TargetObject orbTargetObject =
       MetaSpace.getTargetObject("ORBRegister");
HashMap orbSkeletonHashtable = orbTargetObject.skeletonHashtable;

//update ORB Skeleton Hash table with new meta object in ORB
if ( OldMO.toString().equals(metaID) ){
    skeleton.setSharable(mobj);
}
// Notify the base objects at base level
MetaSpace.notifyBaseObject();
```

In replacing meta objects at runtime, the procedure is as follows:

1. Kernel Manager first instantiates the new meta object with `newInstance()` by using a given meta object name MetaName which is then attached to meta space. At the same time, Kernel Manager constructs the `TargetObject`, which is the connector (a handoff box) mentioned in Sections 11.2 and 11.3 when the meta object is attached to meta space. Using `TargetObject` in meta space is to avoid the cohesive dependency between meta objects. (Step 1)

2. Kernel Manager verifies existence of the old meta object in meta space and uses MetaID and MetaType of the two meta objects to compare and ensure that they have the same meta type. For example, `metaFixedThreadPool` and `metaGrowthThreadPool` have the same identity `ThreadPool` and belong to the same meta type `Pooling`. (Steps 2, 3).

3. Before the new meta object replaces the old meta object, the Kernel Manager transfers the information in `TargetObject` such as system table or port no of the old meta object to the `TargetObject` of the new meta object. (Steps 4, 5, 8, 9, 10).

4. In between the replacement of meta objects, Kernel Manager detaches the old meta object from the meta repository and releases all its resources (stops threads), and attaches the new meta object to the meta repository and claims all needed resources (starts threads) . (Steps 6, 7).

5. For ORB object distribution, Kernel Manager rebinds the new meta object in (ORB) `Registry`. (Step 11).

6. `ORBRegister` has a hash table which contains all skeletons of meta objects in meta space. A `Skeleton` in ORB is responsible for calling methods of *Callee*

objects (meta objects in MELC) on behalf of remote clients. Kernel Manager refreshes the skeleton hash table in the `ORBRegister` with the `skeleton` of new meta object. (Steps 12, 13).

7. At the end, Kernel Manager notifies the base objects (`E-Bookshop`) that have reified the replaced (old) meta object to accept the new meta object for performing object reflection in the architecture. (Steps 14, 15).

Kernel Manager instantiates a new meta object with the method `newInstance()`, where `newClassName` stores the class name of the new meta object. For example, the new meta object *Growth Enabled Thread Pool* has the Java class name `metaGrowableThreadPool.class` to replace the old meta object *Fixed Thread Pool* which has the Java class name `metaFixedThreadPool.class`. For meeting the requirements of meta objects replacement, they both must have same MetaID and MetaType. In this case, they both are `ThreadPool` and `Pooling` respectively. The old meta object is detached from meta repository before the new one attached. Meanwhile, the concerned instances (*skeleton table* and *port number*) of the old meta object are transferred to the new one. After that, the new meta object (`metaGrowableThreadPool`) registers in `ORBRegistry` and updates ORB skeleton table respectively. Lastly, kernel manager informs the base objects (`E-Bookshop`, `E-Banking`) that had reified the meta object (`ThreadPool`) about the changes. Henceforth, the `metaGrowableThreadPool` can continue to serve `E-Bookshop` and `E-Banking` on behalf of the old meta object `metaFixedThreadPool`.

Appendix D – Distributed Computing Technologies

D.1 Distributed Computing Applications

Our framework aims at providing services for distributed enterprise applications. A distributed system is a combination of several computers with separate memories linked over a network, on which it is possible to run a distributed application. A distributed application is an application which consists of several parts of a program communicating with each other, and cooperating to carry out a common task [29]. Typically, but not necessarily, the parts of the application are distributed across several computers. The distribution can also be simulated on one computer. In this case, however, information is not transmitted via a common memory or address space, but with the aid of the techniques of remote communication.

On the lowest level, we find the mechanism of transmitting data streams *via sockets*. Data is transferred from one computer to another.

From the programmer's point of view, it is preferable not to have to communicate between computers, but between objects, as happens in object-oriented programming. The enabling mechanism is RMI, the *Remote Method Invocation*. However, it works only when JAVA is used on both sides, which means that both objects must be expressed in the same language. Further, each object must know the other's location.

If one abstracts further and wants to conceal the location and language of a remote object, one arrives at CORBA, the *Common Object Request Broker Architecture*. Here, the programmer can communicate with the remote object

without knowing where it is and in which language it has been implemented [30].

D.2 Network Services - Transmission Protocols

The basic prerequisite for programming in distributed systems is the ability to transfer data from one part of a system to another. From the programmer's point of view, such data is provided as bits and bytes, as ASCII characters, or even as objects and should be transferred from one computer to another. For the transmission, physical media such as copper wires or fiber optic cables are available over which electrical signals are sent and received.

The gap between the representation of data at the programming level and at the physical level is closed by protocols, a kind of linguistic convention between the individual devices involved in data transmission. The protocol family which is used on the Internet and which combines the different protocols of the sub-networks it consists of is UDP and TCP/IP.

User Datagram Protocol (UDP) [28] is a connectionless transport protocol, which means that it doesn't guarantee packet delivery nor packet arrival in sequential order. Rather than reading from, and writing to, an ordered sequence of bytes, bytes of data are grouped together in discrete packets, which are sent over the network. The packets may travel along different paths, as selected by the various network routers that distribute traffic flow are generally unpredictable, which are depending on factors such as network congestion and priority of routes.

Transmission Control Protocol (TCP) [28] is a stream-based method of network communication. TCP provides an interface to network communications that is radically different from the User Datagram Protocol. TCP uses a lower-level communications protocol, the Internet Protocol (IP), to establish connection

between machines. This connection provides an interface that allows streams of bytes to be sent and received, and transparently converts the data into IP datagram packets. The virtual connection between two machines is represented by a *socket*.

Sockets [30] allow data to be sent and received; there are substantial differences between a UDP socket and a TCP socket. First, TCP sockets are connected to a single machine, whereas UDP sockets may transmit or receive data from multiple machines. Second, UDP sockets only send and receive packets of data, whereas TCP allows transmission of data through byte streams represented as an *InputStream* and *OutputStream* in JAVA.

Data transmission over TCP streams is said to be reliable [27]. UDP is not reliable. Delivery of data is guaranteed by the TCP: data packets lost in transit are retransmitted. Each datagram packet of a TCP socket contains a sequenced number that is used to order data. Packets arriving before earlier packets will be held in a queue until an ordered sequence of data is available.

TCP guarantees that the data packets actually arrive, and even do so in the right sequence. As the administrative effort that is required for TCP is not required for UDP, the latter gains a bit of speed, yet the process is less suitable for transmitting files, as then every bit matters.

D.3 Distributed Communication

All communication between subsystems in a distributed environment must be restricted to messages. Tasks in different subsystems may communicate with each other using several different types of message communication described as follows:

Asynchronous Message Communication

Asynchronous (loosely coupled) message communication is by means of message queues. In distributed environments, loosely coupled message communication is used wherever possible for greater flexibility. The producer task sends a message to the consumer task and does not wait for a reply. The two tasks proceed asynchronously, and a message queue might build up between them. Group communication, where the same message is sent from a source task to all destination tasks that are members of the group, is also supported.

Synchronous Message Communication

Synchronous (tightly coupled) message communication is in the form of either single-client/server communication or multiple-client/server communication. In both cases, a client sends a message to the server and waits for a response; in the latter case, a queue might build up at the server. The producer task sends a message to the consumer task and then waits for a reply from the consumer. The response might be a negative acknowledgement, indicating that the destination node did not receive the message.

D.4 Distributed Object Broker Communication

In a distributed object environment, clients and servers are designed as distributed objects. An object broker is an intermediary in interactions between clients and servers. It is transparent and frees clients from having to maintain information about where a particular service is provided and how to obtain that service. It provides location transparency, so that if the server object is moved to a different location, only the object broker needs to be notified. We consider transparency in object distribution is one of important properties for distributed computing application development.

The interaction is shown in Figure D.4[29] in which the dialogue is typically:

1. Client sends a request to the Broker.
2. The Broker looks up the location of the server and returns a service handle to the Client.
3. The client uses the service handle to request the service from the appropriate Server.
4. The Server services the request and sends the reply directly to the Client.

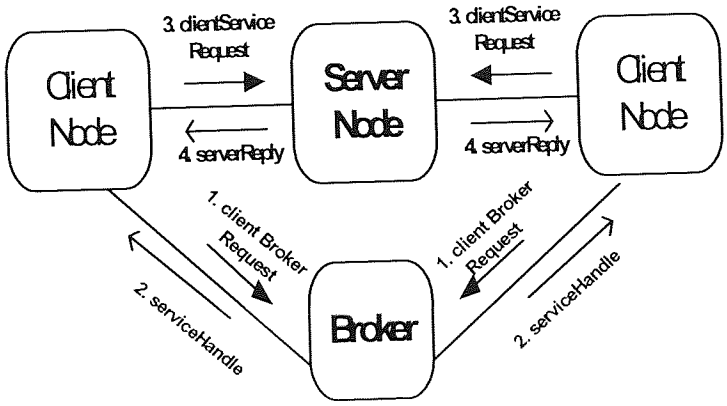

Figure D.4 Interaction for Object Broker Architecture

Common Object Request Broker Architecture (CORBA) uses this approach which is efficient if the client and server are likely to have a dialog that results in the exchange of several messages. With this approach, it is the responsibility of the client to discard the handle after the dialog is over. CORBA is a distributed framework designed for support of heterogeneous architectures [31]. While Java's RMI provides support for homogeneous architecture, with Java at both ends and wiring in the middle, CORBA allows for connection between two different and therefore heterogeneous systems [32]. Such systems may differ not only by the hardware they use, but also by their operating system and programming language.

D.5 Remote Method Invocation (RMI)

Remote Method invocation (RMI) [33] is one of the cornerstones of Enterprise JavaBeans and is an extremely handy way to make distributed Java applications. Instead of invoking a method on another Java object running in the same Java Virtual Machine, you invoke a method in a Java object in another JVM on the same computer or a different one.

RMI is virtually seamless. Users don't have to do much to enable a class for RMI. RMI revolves around the use of *Remote interface* that defines all the remote methods for an object. RMI creates an object called a *stub* that implements the Remote interface and runs in the client's JVM. When the client invokes a remote method, the stub's implementation of the method results in transmitting the method invocation over to the server's JVM where another special object called a *skeleton* interprets the request and invokes the correct method. When the server's method returns a value or throws an exception, the skeleton packages the resulting information and sends it back to the stub. The stub then returns the information to the client. RMI Architecture is shown in Figure D.5.

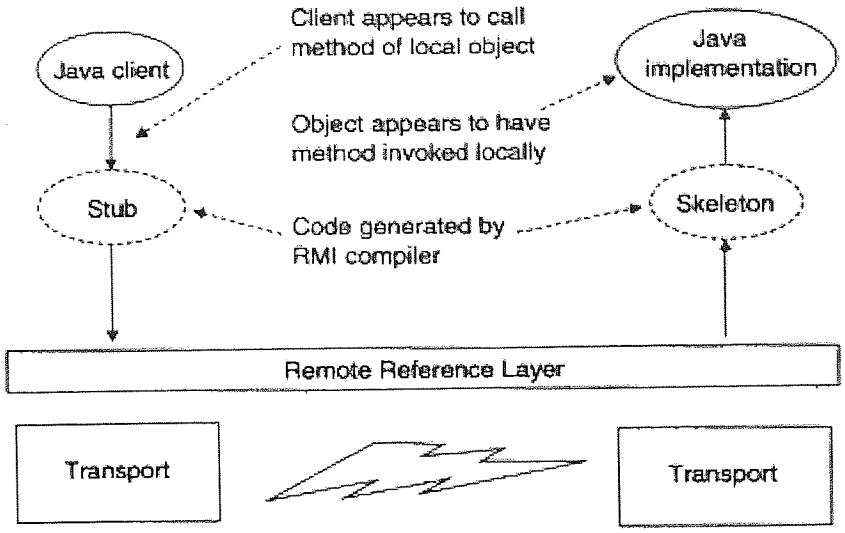

Figure D.5 The RMI Architecture

The nice thing about the stub is that the client does not know that it is talking to a stub. The client thinks it is just invoking a method through an interface. Likewise, the server doesn't know about the skeleton at all. As far as the server is concerned, the skeleton is just like any other class that uses the server class [13].

D.6 Common Object Request Broker Architecture (CORBA)

CORBA is a distributed framework designed for support of heterogeneous architectures, which allows for connection between two different and therefore heterogeneous systems [31]. Java IDL provides an implementation of the CORBA 2.0 specification.

CORBA Components

A CORBA implementation consists of several pieces, Object Request Broker (ORB), Interface Definition Language (IDL) compiler, one of more implementations of Common Object Services (COS), also known as CORBAServices and Common Frameworks, also known as CORBAFacilities.

programming language, which is neutral to any other programming language but can transform itself with IDL compiler into any other computer language in defining the provision of a remote service.

Internet Inter-ORB Protocol

The Internet Inter-ORB Protocol is a TCP/IP implementation of the General InterORB Protocol (GIOP). With CORBA it is possible to build a client application using one vendor's ORB and IDL compiler, build a server or object implementation with a second vendor's ORB and IDL compiler, and create a set of common services for both client and server with yet a third vendor's ORB and IDL compiler. IIOP allows each of the three different vendor's products to communicate with each other using a standard set of protocol semantics.

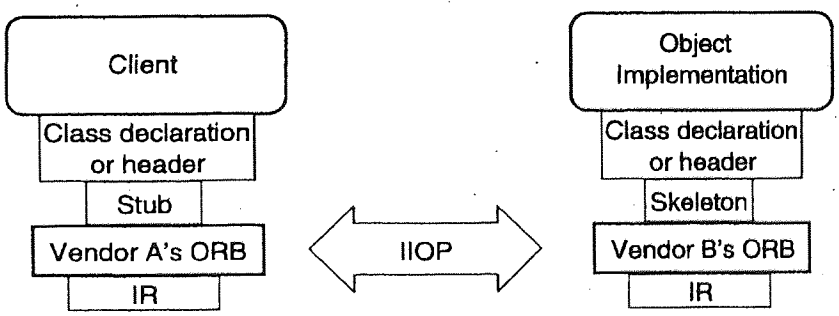

Figure D.6 The CORBA Architecture

And, when users consider that all three of these ORBs could be using different programming languages and running on different hardware and operating system platforms, the architecture that CORBA has is rather flexible in implementation.

D.7 Summary

In distributed computing, application developers design the communications between the client and server, which decide how a client actually sends a message to a remote object. The three most popular solutions are the Common Object Request Broker Architecture (CORBA), Remote Method Invocation (RMI), and custom sockets.

The advantage of CORBA is that it defines a number of services that users frequently need in a distributed object system. CORBA has standards for naming and events. CORBA is language-neutral. A client written in Java can communicate with a remote object written in C++. Custom socket solutions are popular because systems such as CORBA are often overkills for small applications. RMI is a Java-only solution. Because it is Java-only solution, RMI can take advantage of all Java's features and ORB (Object Request Broken), and it hides the object distribution, which is transparent for developing object distribution for Java developers. Because of transparency, simplicity and popularity of RMI, Sun chose RMI (Internet Inter-ORB Protocol) as the framework for communicating with Enterprise JavaBeans and is the cornerstone of Enterprise JavaBeans (EJB). RMI has gradually been edging out CORBA as the preferred technology for distributed object development [33]. In fact, we employ RMI (ORB Protocol) as middleware in our framework for distributed objects communication also because of its *transparency* in object distribution, which meets one of major properties in our adaptable distributed framework.