Aspect-Oriented Refactoring to Patterns

by

Meenal Nagappan

A thesis submitted to the Faculty of Graduate Studies and Research

in partial fulfillment of the requirements for the degree of

Masters

in Computer Science

Ottawa-Carleton Institute for Computer Science

School Of Computer Science

Carleton University,

1125 Colonel By Drive

Ottawa, Ontario, Canada, K1S 5B6

September, 2005

© 2005, Meenal Nagappan

Library and
Archives Canada

Bibliothèque et
Archives Canada

Published Heritage
Branch

Direction du
Patrimoine de l'édition

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-494-10119-9

Our file Notre référence

ISBN: 0-494-10119-9

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protègent cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

When new commercial software systems are developed, or existing ones are enhanced, non-functional features (such as logging and security) are typically added to the system as an after-thought. To meet market pressure, initial versions of the system are deliberately kept simple and primarily focused on functional requirements. In subsequent versions, non-functional features identified for system enhancement are added to it. The drawback of such an approach is that it makes the initial simple version increasingly more complex, and thus more difficult to support and maintain. The aspect-oriented approach has been proposed to add such non-functional features (commonly referred as “ilities” in the field of computer engineering) to software systems without polluting or affecting the integrity of the original code. In other words, non-functional features implemented in the form of aspects stand isolated without interfering with the simplicity, readability and integrity of the original system. This research brings together solutions to non-functional requirements using patterns and refactoring to patterns in an aspect-oriented approach. The proposed solution has been tested by adding non-functional *ilities* to an existing RMI-based instant messaging system using patterns and aspect-oriented approach. The proposed solution is also tested to refactor an existing non-functional feature. The main proposal of this research is that aspects are a vehicle for adding refactoring/patterns to a system. Re-usable aspect libraries that are designed and implemented as part of this research could be directly integrated into real world applications.

Acknowledgements

I sincerely thank my supervisor, Prof. Michael Weiss for his support, encouragement, guidance, and advice throughout my graduate studies. I would also like to thank my supervisory chair, Prof. Barbeau and committee members, Prof. Amyot and Prof. Esfandiari for reviewing my research work.

I also owe my sincere thanks to all the faculty of the school of Computer Science. I would take this opportunity to thank Linda Pfeiffer, Claire Ryan, Sharmilla Namasivayampillai and Jane McArthur for their extra care. I would also like to thank anonymous reviewers of academic papers, who provided several useful comments, criticisms, and suggestions.

On the personal side, I am grateful to my husband, Ram, who has always been wonderfully supportive, encouraging and loving. I am also thankful to my mother for her selfless care and affection. Finally to my in-laws, grandfather and friends, for their support.

Dedication

To

My Late Father,

Mr. Subramanian Nagappan

Table of Contents

ABSTRACT	II
ACKNOWLEDGEMENTS.....	III
DEDICATION.....	IV
TABLE OF CONTENTS	V
LIST OF FIGURES	VIII
LIST OF TABLES.....	IX
LIST OF ACRONYMS AND ABBREVIATIONS	X
GLOSSARY OF IMPORTANT TERMS.....	XI
1. INTRODUCTION	1
1.1. Motivation.....	1
1.2. Problem Statement.....	2
1.3. Proposed Solution.....	3
1.4. Thesis Contribution	5
1.5. Organization of the Thesis.....	6
2. BACKGROUND	8
2.1. Non-Functional Requirements	8
2.2. Aspect-Oriented Programming.....	8
2.3. Refactoring Techniques	13
2.4. Patterns	14
2.4.1. <i>GoF Patterns</i>	15
2.4.2. <i>Concurrency Patterns</i>	16
2.4.3. <i>Applied Patterns</i>	17

2.5.	Summary	23
3.	GENERAL APPROACH	25
4.	CASE STUDY -- INSTANT MESSENGER APPLICATION	27
4.1.	Architecture of UIM	27
4.2.	UIM Scenarios	28
4.2.1.	Registration	28
4.2.2.	Cancellation	28
4.2.3.	Login	29
4.2.4.	Logout	29
4.2.5.	Add Friend	29
4.2.6.	Remove Friend	29
4.2.7.	Send Message	30
4.2.8.	Change Status	30
4.3.	Design of UIM before enhancement	30
4.3.1.	Classes and their Roles	30
4.3.2.	Feature Deficiencies	31
4.4.	Design of UIM after enhancement	35
4.4.1.	Maintainability Enhancement	35
4.4.2.	Serviceability Enhancement	35
4.4.3.	Accountability Enhancement	36
4.4.4.	Security Enhancement	36
4.4.5.	Scalability Enhancement	37
4.4.6.	Durability Enhancement	37
4.5.	Class Diagram of UIM After Enhancement	37
4.6.	Enhanced User Interface of UIM	37
4.7.	Summary	39
5.	IMPLEMENTATION	40
5.1.	Collaborator Mediator Aspect	41
5.2.	Decorator Wrapper Aspect	46
5.3.	Code Locking Aspect	49
5.4.	State Aspect	52
5.4.1.	Comparison of the four general approaches to address security	59
5.5.	Thread Pool Aspect	60
5.6.	Delayed Retry Aspect	66
5.7.	Integration of Aspects	69

5.8.	Summary	70
6.	EVALUATION	71
6.1.	Comparison Evaluation of Conventional and Proposed Solution.....	71
6.1.1.	<i>Advantages</i>	72
6.1.2.	<i>Limitations</i>	78
7.	CONCLUSIONS AND FUTURE WORK.....	80
7.1.	Conclusions	80
7.2.	Future Work.....	81
	REFERENCES.....	82
	APPENDIX A ASPECTJ V1.1 KEYWORDS (QUICK REFERENCE)	86
	APPENDIX B ADDITIONAL CLASSES AND ASPECTS FOR COLLABORATOR MEDIATOR PATTERN	88
B.1	RMI implementation of Server in OOP	88
B.2	Collaborator mediator aspect for Agent	91
	APPENDIX C CLASSES REFERRED IN THREAD POOL ASPECT	94
C.1	Class - AsyncClientCallBack.....	94
C.2	Interface – ClientUpdate	96
C.3	Class – RequestClientAsyncCallBack	96
C.4	Class – NotifyThread	96
	APPENDIX D LIST OF COMMON POINTCUTS	99

List of Figures

FIGURE 2.1: AN EXAMPLE ASPECT	11
FIGURE 2.2: AN EXAMPLE CLASS.....	11
FIGURE 2.3: A DISCREET LOGGING ASPECT	13
FIGURE 3.1: OPTIONS FOR ADDRESSING NON-FUNCTIONAL FEATURES.....	26
FIGURE 4.1: ARCHITECTURE OF UIM	27
FIGURE 4.2 CLASS DIAGRAM OF UIM BEFORE ENHANCEMENT	34
FIGURE 4.3 CLASS DIAGRAM OF UIM AFTER ENHANCEMENT	38
FIGURE 4.4: USER INTERFACE OF UIM.....	39
FIGURE 5.1: COLLABORATORS AND A MEDIATOR.....	41
FIGURE 5.2: DECORATOR WRAPPER ASPECT FOR WRITING.....	47
FIGURE 5.3: DECORATOR WRAPPER ASPECT FOR READING	48
FIGURE 5.4: ABSTRACT CODE LOCKING ASPECT	50
FIGURE 5.5: CONCRETE CODE LOCKING ASPECT	50
FIGURE 5.6: STATE ASPECT	52
FIGURE 5.7: CLASS – PBEENCRYPTER WITH 64-BIT KEY.....	53
FIGURE 5.8: APPROACH WITHOUT USING PATTERNS AND ASPECTS	54
FIGURE 5.9: APPROACH USING ONLY PATTERNS	55
FIGURE 5.10: APPROACH USING ONLY ASPECTS	56
FIGURE 5.11: PRIMARY CODE CAPTURED BY THE PATTERNS AND ASPECTS APPROACH....	57
FIGURE 5.12: SEQUENCE DIAGRAM FOR UPDATE STATUS PROCESS AS CONCEIVED IN THE ORIGINAL VERSION OF UIM	61
FIGURE 5.13: THREAD POOL ASPECT	62
FIGURE 5.14: MODIFIED SEQUENCE DIAGRAM FOR UPDATE STATUS PROCESS IN THE ENHANCED VERSION OF UIM	65
FIGURE 5.15: ABSTRACT DELAYED RETRY ASPECT	67
FIGURE 5.16: CONCRETE ASPECT INHERITING DELAYED RETRY ASPECT	68
FIGURE 5.17: REGISTER METHOD CAPTURED BY DELAYED RETRY ASPECT.....	69
FIGURE 5.18: SET PRECEDENCE FOR ASPECTS	69

List of Tables

TABLE 5.1: COLLABORATOR MEDIATOR ASPECT-1 42

List of Acronyms and Abbreviations

AO	- Aspect-Oriented
AOP	- Aspect-Oriented Programming
API	- Application Programming Interface
BEB	- Binary Exponential Back-off
CORBA	- Common Object Request Broker Architecture
GoF	- Gang of Four
GUI	- Graphical User Interface
FIFO	- First In First Out
IDE	- Integrated Development Environment
IEEE	- Institute of Electrical and Electronic Engineers
JVM	- Java Virtual Machine
OO	- Object-Oriented
OOP	- Object-Oriented Programming
POSA	- Pattern-Oriented Software Architecture
RMI	- Remote Method Invocation
UIM	- Universal Instant Messenger

Glossary Of Important Terms

In this glossary, the meanings of some terms that are used frequently are summarized.

The number in round parentheses refers to the section where the left-side term is defined.

Aspect is a code snippet written to implement a non-functional feature. It is separated from the main system code [6]. (Section 1)

-ilities are non-functional features in a computing system and most of them end with a “-ility”[7]. (Section 1)

Pattern is an accepted/proposed solution for addressing a recurring design [15]. (Section 1)

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior [21]. (Section 1)

1. Introduction

1.1. Motivation

Industrial personnel and academic researchers face many challenges when a software system grows and has to continuously adapt to changes in the environment. When a software system grows without consideration of its structure, it becomes very hard to understand, extend and maintain it. In this highly competitive market, software extensibility (evolution) is vital for a system to survive. Conventional (object-oriented) software engineering offers many different ways to address the design evolution. Within this framework, a few options available are:

1. Rewrite a partial or entire system to handle unanticipated evolutions.
2. Use best practices (a set of guidelines).
3. Apply refactoring techniques.
4. Apply design patterns.
5. Combine one or more of the practices mentioned above.

Motivations for the current research are derived from the work done in the areas of: aspect-oriented programming, refactoring techniques and design patterns.

1. *AspectJ in Action* [31] is a pioneer textbook on Aspect-oriented programming. This book simplifies many real world problems using aspect-oriented solutions based on complex conventional programming approaches.

2. Typically, aspect-oriented refactoring [9] helps to shrink the complex code in length thereby improving the understandability and maintainability of the code.
3. *Refactoring to Patterns* [21] is a textbook combining both refactoring techniques [6] and design patterns [15] in a single package. This textbook using two of the techniques offers a valuable solution for many design issues. In this textbook, each code refactoring is done on a particular pattern, i.e. design issues are turned towards the direction of patterns. It states, there is not just one way to implement a pattern, i.e. it does not emphasize on a particular way of implementing patterns. In addition, it states “refactoring to patterns makes the code clean and easy to follow due to the familiarity of patterns that most programmers possess. Therefore, the current research is interested in evaluating aspect-oriented refactorings to pattern.
4. *Pattern-Oriented Software Architecture* [13] discusses many concurrent pattern-based solutions, which are readily applicable to problems in distributed and concurrent systems.

1.2. Problem Statement

Systems continuously evolve to include advanced features or to improve performance. The reasons for this continuous evolution are multifold. However, many of these advanced features are typically non-functional in nature. Such features are not the primary functionality of the system but are essential for the system to work efficiently and reliably. For example, logging is considered an essential non-functional feature in complex

applications to make the system more maintainable. Some of the popular and conventional Object-Oriented Programming¹ (OOP) way of addressing the continuous evolution escalates the complexity of the system.

The reasons and the results of the system complexity are as follows:

- Typically, non-functional features are implemented in a redundant and intertwined fashion throughout the system.
- Therefore, the understandability of such system becomes harder.
- Non-functional features implemented in the above-described manner are not readily reusable.

1.3. Proposed Solution

This research states that the described above problems could be handled efficiently by refactoring to patterns using aspects. Therefore, the research mainly focuses on a combination of the aspect-oriented approach with the following two design techniques:

- Refactoring techniques
- Patterns

¹ Popular OOP hybrid languages such as Java, C++, C#, etc.

Because aspects can introduce non-functional features without polluting the original system code, they are becoming increasingly popular. In [11], persistence is added as an essential non-functional feature to a bibliographic application using aspects. In [12], refactoring a CORBA [2] middle-ware implementation, using aspects, is analyzed. Weaving contextual collaboration as aspects into a legacy application such as an address book has been explored in [26]. In [18] and [32], aspects are used at the model level. Aspect-Oriented Programming (AOP) can be considered as a further enhancement of the OOP. OOP modularizes software procedures into classes and methods. In OOP, it is difficult to modularize concerns (e.g. logging and security) spanning from several classes and methods. Therefore, this leaves the implementation of the concern, scattered throughout the system, and tightly intertwined with the core system functionality. AOP provides aspects as a construct for separating and modularizing concerns and helps prevent code tangling. A comprehensive introduction to AOP is provided in [31].

A definition of refactoring [6] states, "Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior". Refactoring increases the integrity and testing phases and decreases the developmental phase of the further releases, thereby improving system quality. Conventional object-oriented (OO) refactoring restructures a system into a cleaner and more understandable one. Whereas aspect-oriented (AO) refactoring supplements the conventional OO refactoring by eliminating scattered and intertwined code thus further reducing the system complexity.

Patterns provide well-defined constructs to recurring design issues. By using patterns, design problems can be often solved quickly, in a proven fashion. Most literature for e.g. [21] assumes that patterns are added to refactor existing functionalities, but this research analyzes adding them for new functionalities (non-functional features). In addition, re-engineering non-functional features using pattern based Aspect-oriented approach enhances the refactoring to patterns approach [21].

In [16], non-functional features are implemented using only aspects approach. But, the current research provides a pattern-based aspects approach for implementing non-functional features in an efficient manner. The advantages of the latter approach over the former are presented in detail in section 5.4.1.

1.4. Thesis Contribution

The contribution of this thesis includes:

- Using the proposed solution to refactoring an existing non-functional feature in a real-world system from a redundant or intertwined manner into a modular manner, illustrated in detail in section 5.1
- Using the proposed solution to directly implement many new non-functional features in a real-world system that otherwise would be intertwined and/or redundant, illustrated in detail in sections 5.3 and 5.6.

- Using the proposed solution to design and create many non-functional features in reusable pattern-based aspect libraries that could be applied in similar situations, illustrated in detail in sections 5.2, 5.4 and 5.5.
- Highlighting the comparison between the conventional OO² approach and the proposed approach, illustrated in detail in section 6.
- Proposed a new pattern, Delayed Retry Pattern to retry a failed action repeatedly in delayed time intervals. The pattern is presented in section 2.4.3.
- The primary focus of this research sheds light on enhancing non-functional features to an existing real world system in a simple manner (i.e. without escalating system complexity due to the enhancements).

The case study used for this research is an existing RMI-based instant messenger system that has been implemented using OO methodology.

1.5. Organization of the Thesis

In chapter 1, a general introduction of the research is presented. Chapter 2 presents background information of the aspect-oriented approach, refactoring techniques and design

² The current research refers conventional Object-Oriented languages to hybrid and popular languages like C++, Java, C# and excludes the ones that support meta-classes like SmallTalk.

patterns. This chapter also describes six patterns that are applied to this research. Chapter 3 compares various approaches to solve non-functional features. Chapter 4 introduces the Universal Instant Messenger (UIM) [8] application used as a case study for this approach. Chapter 5 describes the proposed way of implementing the design patterns identified in chapter 3. Chapter 6 evaluates³ the proposed approach and describes its advantages and disadvantages. Finally, chapter 7 includes suggestions for future research and concludes the research.

³ Eventhough, the evaluation is based on the case study (discussed in section 4), results are not specific to the case study; can be extended for other systems as well.

2. Background

This chapter provides background information on non-functional requirements on which the proposed solution is applied. In addition, it presents background information on following concepts applied to the proposed solution: aspect-oriented programming, refactoring techniques and patterns.

2.1. Non-Functional Requirements

Deducing requirements, which is the first step of a system lifecycle consists of two categories: functional and non-functional. Functional requirements are associated with specific functionalities that a system is expected to possess. Whereas non-functional requirements supports/enhances the functional requirements such that the quality of the system is improved. The non-functional requirements are referred as “ilities” (most of them end with –ility) in a computing system. Consider a Banking system, where making deposits/withdrawals are associated with functional requirements (features) while enhancing security in deposits/withdrawals are associated with non-functional features.

2.2. Aspect-Oriented Programming

Because OOP emphasize on reusability and modularization of features, the methodology is becoming the mainstay for software development. However, there are limitations in its abstraction and composition mechanisms. Using it to totally isolate the essential non-functional features (concerns) from core functional features of a system is difficult. In

other words, it does not support mechanisms to add non-functional features without polluting the code of a primary system. For example, in developing any system, exception handling is not a functional concern, but it is typically scattered with primary functionalities. Thus, in OOP within a single module, several non-functional features are tightly intertwined with the primary system code in contradiction to its primary objectives, modularization and reusability. Because of its limitations, it also makes the primary code messy and becomes difficult to enhance or maintain the system.

AOP complements OOP by introducing aspects. An aspect is a new type of abstraction and composition mechanism for introducing non-functional features to a system without polluting its primary system code. In other words, aspects can be used to modularize non-functional features that otherwise tend to scatter across primary features. By isolating non-functional features from the primary functional features, AOP greatly enhances the possibility of reusing both functional and non-functional modules.

In a primary system, there are mainly two situations to apply aspects:

- When a same piece of code is scattered along many places.
- While modules instead of addressing, a single non-functional feature contains intertwined code of additional features.

AspectJ [1], AspectC++, and AspectC# are some of the AOP languages. Respectively AspectJ, AspectC++, and AspectC# enhance Java, C++ and C# object-oriented languages. Typically, in the context of AspectJ (a list of AspectJ keywords is provided in

Appendix A) an aspect consists of terminologies such as join points, pointcuts and advice. These terminologies are referred by a different keyword in different AOP languages; however, their basic ideas remain the same. The terminologies are defined in detail in [31] and are summarized as follows:

Join points: Identifiable points at the time of system's execution are referred as join points. Some of the examples for join points are call to a method, execution of a method, object construction, a conditional check, and exception handler. Join points are of two types: absolute and collective. An example of absolute join point is `Class1.method1`; which refers to the `method1` of `Class1`. An example of collective join point is `Class1.method*`, which by using the wild card `*` refers to all the methods of `Class1` whose name begins with `method`.

Pointcuts: A pointcut is used to select join points and collect context at the selected join points. A pointcut for selecting the join points `Class1.method*` can be stated as,

```
pointcut pointcut1(): call (public void Class1.method*());
```

A list of common pointcuts applied in this research is provided in Appendix D.

Advice: Advice is a piece of code to be executed at a join point that has been selected by a pointcut.

There are three types of advices: before, after and around. A before advice executes before the normal execution of a captured join point. An after advice executes after the normal execution of the captured join point. An around advice manipulates the normal execution of the captured join point. The three types of advices are illustrated in Figure 2.1, in the context of a complete example aspect. Figure 2.2, illustrates an example class to which the advice in Figure 2.1 are applied.

```
public aspect Asp1 {
    pointcut pointcut1():
        call (public void Class1.method*());
    before():pointcut1 (){
        System.out.println("Hi! from call pointcut");
    }
    void around():pointcut1 (){
        System.out.println("This around()
            makes the captured method do
            nothing and just prints this message");
    }
    after():pointcut1 (){
        System.out.println("Bye! from call pointcut");
    }
} // Eof Asp1
```

Figure 2.1: An Example Aspect

```
public class Class1 {
    private static int x=5, y=5, sum = 0, diff = 0;
    public static void method1(){
        sum = x + y;
    }
    public static void method2(){
        diff = x - y;
    }
    public static void main(String[] args) {
        Class1.method1();
        Class1.method2();
    }
} // Eof class
```

Figure 2.2: An Example Class

As aspects are implemented as separate entities (for non-functional features) in isolation from the primary system code, they are integrated with the primary system either at compile-time or at run-time. This process of integrating the aspects with the primary system is called aspect weaving. Compile-time weaving uses an aspect compiler for source-to-source translation (source code for primary features and non-functional features, the aspects) for producing woven source code. The woven source code is then processed by the base language compiler to produce final executable binaries. Run-time weaving dynamically incorporates the aspects with the targeted (primary) system code during execution. Aspects could be abstract or concrete. An aspect is declared abstract, if any of its pointcuts are abstract. Abstract aspects enable instant (ready-made) reuse within or outside the system's scope. Concrete aspects provide implementation specific pointcuts that captures specific join points in the primary system. Thus, when reusing concrete aspects, the pointcuts have to be modified appropriately.

Because in AOP, aspects can be added to a system discreetly (an example for such a discreet enhancement is provided in Figure 2.3) without the knowledge of primary functional modules (or with as little involvement of primary functional modules), non-functional features can be added to a system as an after-thought. This obliviousness decouples non-functional features from the primary functional code, which makes it easier to maintain complex software systems. Thus using AOP increases the efficiency of the overall software development process.

Figure 2.3: A discreet logging aspect

2.3. Refactoring Techniques

As discussed earlier, refactoring is a technique that changes the intrinsic building blocks of a system without changing its external conduct. This change is done to improve the system's readability and understandability. In practice, refactoring is achieved by a step-by-step system refinement so that any errors introduced due to refactoring could be captured and fixed along the steps. Fowler [6] is a pioneer in the field of refactoring techniques. A catalog of refactoring is also detailed in [6]. A basic example of the refactoring-technique is described briefly in the following paragraph.

Rename Method Refactoring: When the name of the method fails to convey its purpose, it is recommended to use the Rename Method Refactoring. For example, if the purpose of *retCon* method is to retry connection to a resource (say, a database), the refactoring-technique suggests renaming the method name to reveal its purpose. Obviously, renaming from *retCon* to *retryDataBaseConnection* would be more meaningful.

As refactoring-technique promotes a better software design and implementation, this in conjunction with aspect-oriented programming improves a system twofold. Object-oriented refactoring techniques are becoming increasingly popular. As evidence, many Integrated Development Environments (IDEs) are built with refactoring techniques such as move and rename. A move facilitates moving a module from one location to another; thereby updating this change to wherever there have been references within the project scope. One aim of this thesis is to analyze how aspect-oriented programming can be used efficiently to implement such refactorings.

2.4. Patterns

Patterns are ready-made solutions (procedures) to recurring design problems in real-world application development [15]. In this perspective, recurring issues refer to the same problems faced by a pattern author as well as others repeatedly. Therefore, pattern authors share their devised solution for such recurring design problems with others to reduce the time spent for designing. A pattern does not require any external tools (software)

to reap its benefits. They encourage code reuse and, thus, support rapid application development. Using patterns result in highly cohesive modules with minimal coupling. The other advantages of using well-known patterns are, they provide a common vocabulary among developers, and simplify the maintenance of application code.

When the refactoring-technique and aspect-oriented programming are brought together with patterns, a system becomes more structured. However, it should be noted that patterns must be used parsimoniously. Overuse of patterns can actually result in designs that are more difficult to understand, and maintain. In addition, they are not always recommended to be used up-front in a design. In many situations, they should instead be introduced as motivated by specific needs, often after portions of the system have been implemented [21]. Hundreds of design patterns are available in the field of computer engineering.

There are many design patterns commonly being used in computing field. This research sheds light on the Gang of Four (GoF) and concurrency patterns. Other such patterns are testability [23] and persistence [27]. A brief overview of these patterns is described below:

2.4.1. *GoF Patterns*

GoF patterns' [15] authors are pioneers in the world of design patterns. These patterns are often described as a starting kit to learn about patterns. They are categorized into three

types: Creational, Behavioral and Structural design patterns. The target audiences of these patterns are object-oriented systems.

Creational patterns are applied to create instances of an object in an efficient manner. Five patterns in this category are: They are Abstract factory, Builder, Factory method, Prototype and Singleton.

Behavioral patterns are generally applied in situations when objects need to communicate with each other. Eleven patterns in this category are: Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template method and Visitor.

Structural patterns are applied to formalize relationship between two entities. Seven patterns in this category are: Adapter, Bridge, Composite, Decorator, Facade, Flyweight and Proxy.

2.4.2. *Concurrency Patterns*

Concurrency refers to simultaneous execution of threads within the same application. Concurrency patterns are well defined and analyzed solutions to prevent discrepancies and to enhance performance when multiple clients or processes simultaneously access a given resource. Implementing concurrency patterns in OOP are discussed in [14]. However, existing approaches only consider the up-front use of concurrency patterns. The in-

tegration of these patterns after (portions of) the system have been designed, presents an unresolved challenge.

Concurrency solutions are essential in an environment where multiple processes compete for shared resources. Such examples are abundant in as the context of multi-threaded programming, distributed computing and networking. Some of the well-defined and widely analyzed concurrency patterns are Half-Sync/Half-Async, Leader/Followers, Active Object, Thread-per Request, Thread-per Session, Thread Pool [13], Shared Resource [22], Code Locking [29], Rendezvous [17], and Asynchronous Completion Token [19].

For the present research, three of the GoF patterns — Collaborator Mediator, Decorator Wrapper and State — and two of the concurrency patterns—Thread Pool and Code Locking—are considered. In addition, a concurrency pattern—Delayed Retry—is proposed. The above-mentioned 6 patterns applied in this research are derived from a case study (discussed in section 4) for addressing/enhancing specific non-functional requirements; they have not been chosen randomly.

2.4.3. *Applied Patterns*

The behavioral (Collaborator Mediator and State), structural (Decorator Wrapper) and concurrency patterns (Code Locking, Thread Pool and Delayed Retry) considered for this research to use in conjunction with aspects are described in the following section.

Collaborator Mediator Pattern

Context: With the set of objects, if each one of them is tightly coupled, the cohesiveness makes it complicated when changes are required. Such a situation promotes reorganizing the objects in a way that minimizes the coupling and facilitates communication among them using one intermediary object.

Intent: The Collaborator Mediator pattern separates objects so that they interact with each other through the mediator and not directly.

Problem: When two or more objects interact with each other without a mediator, results in tight coupling. Even a small change in behavior needs change(s) in several objects referring it. If the system is distributed, such changes to tightly coupled objects require more work, as the modifications have to be distributed as well. Therefore, the problem requires handling interaction of two or more objects in a separable fashion.

Forces:

- A set of related objects tend to involve in multiple dependency relationships due to tight coupling.
- These objects become difficult to reuse/modify as their functionality is interlinked with dependency relationship.

Solution: The Collaborator Mediator pattern promotes loose coupling by keeping objects from referring to one another explicitly as the interactions are separated into a mediator object. The mediator object does the interactions and the other objects are loosely coupled allowing more reusability and easier maintenance.

Decorator Wrapper Pattern

Context: The ability to dynamically wrap one wrapper inside another to enhance or withdraw functionalities.

Intent: This pattern is used instead of object inheritance to minimize maintenance, when the inherited subclasses become complicated.

Problem: Requires enhancing/withdrawing functionality to/from objects without affecting their existing behavior.

Forces: When the inherited subclasses grow more complex, extending functionalities through inheritance is not advisable. The motivating forces for this pattern are explained using an example. Consider a base object, a car. A feature namely an audio can be added to it. This new feature could be very well implemented using inheritance, but if the audio needs to be added to a truck and many different objects, there will be many subclasses, which makes the system (to which the object belongs to) complex. Another and better approach to handle this situation would be to enclose objects (e.g. car and truck) in another object that adds the audio. By default, a car may not have audio in it but could be added as and when required. This enclosing object is called as a decorator.

Solution: The decorator is referred as a wrapper. The decorator adds/removes functionality to/from the caller object. As the decorator adds functionality at runtime, there is no tight coupling between the object and the decorator. Whereas inheritance adds functionality at compile time (due to its static behavior), there is tight coupling between the objects and the functionality. Therefore using decorators in such situations offer flexibility and promote customization in adding/removing/replacing them.

State pattern

Context: State pattern is applied when a particular action causes an object to change its state from one to another (or when an object is initialized for the first time). This action could have been one out of possible options that the object needed to take in order to continue its execution. Basically, the whole system consists of states, their transitions and their action within different states.

Intent: Causes an object to alter its behavior by changing states.

Problem: A situation where an object needs to change its behavior at runtime depending on its state.

Forces:

- An object's behavior depends on its state (i.e. it must change its behavior at runtime depending on one of the possible states).
- When there are many branches in conditional statements, the option for the object is to take one of the many branches.

Solution: Using this pattern, a specific behavior for a state (for e.g. when there are multiple branches in a conditional statement) is separated into modules. As a result, changes to existing states and addition of new states are addressed in each of the state's modules, making the code easier to understand. This pattern allows treating the object's state as an object that can be modified independently from other objects.

Code Locking Pattern

Context: This pattern applies to a commonly encountered situation, the corruption of shared resources as a result of concurrent manipulation.

Intent: The goal of this pattern is to protect portions of code (so called critical sections) responsible for updating a shared resource, and to avoid the corruption of the shared resource by preventing multiple processes from simultaneously accessing the resource.

Problem: Need to give processes uninterrupted access to a critical section (shared resource).

Forces: For a system to be reliable there is a need to prevent corruption of shared resources. On the other hand, restricting access to shared resources may incur a performance penalty, as access is serialized.

Solution: Code locking focuses on locking and unlocking a critical section of code. The solution is to synchronize the access of all processes to the critical section. Typically, critical sections only occupy relatively few lines compared to the full code. Code Locking suggests locking the critical code section just prior to entering the section and releasing immediately after leaving it.

Thread Pool Pattern

Context: This pattern addresses the situation when tasks are executed in their own threads.

Intent: The main objective is to optimize the way the following scenario is handled: a) the number of tasks is too many, b) the tasks are short-lived and c) the response time for addressing individual tasks has to be minimized.

Problem: There is a need to execute more than one task simultaneously in order to enhance performance scalability.

Forces: By having a set of created threads repeatedly available for performing tasks, the pattern optimizes the service time of tasks and minimizes the overhead involved in creating and destroying per requests threads. If threads are created, on a per-request basis, without bound and if a burst of requests arrives, the system resources could be exhausted. Because of resource exhaustion the system can crash.

Solution: A pool of n number of threads is created to perform m number of tasks. Typically n is much smaller than m and the tasks awaiting service are held in a queue. As soon as a thread completes its current task, it requests the next task from the queue until all tasks have been completed. The thread can then sleep until there are new tasks available. The number of threads in the pool can be tuned to provide optimal performance. The advantage of using a thread pool, over creating a new thread for each task, is minimized overhead of creating and destructing threads. This results in better performance and system stability.

Delayed Retry Pattern

Context: This pattern applies to a commonly encountered situation—contention collision—in distributed environments. Contention collision is a common problem in broad-

cast networks. Contention collision occurs if multiple interconnected communication systems transmit simultaneously over a shared access medium.

Intent: The goal of this pattern is to retry contention in a disciplined manner.

Problem: When multiple processes compete for a shared resource, contention collision can occur, when processes immediately retry accessing the resource right after a contention failure. Contention collision also occurs, when communication is attempted with a (temporarily) failed or a busy system.

Forces: Retrying immediately after a collision increases the chance of another collision. Minimizing contention collision is essential to: maximize system resource usage, minimize resource access latency, and to increase system robustness.

Solution: Contention collisions can be minimized if the contention retries are attempted in a disciplined manner. If a requested resource is not available, simple give-up or constant retries should be extended to handle retries at increasingly delayed random intervals. A variation of this solution—the binary exponential back-off (BEB) algorithm—is adopted as a part of IEEE 802 standard. The retry scenario explained here tries repeatedly with an exponential random delay [*sleep_time* = *random* (*INITIAL_DELAY*, *current_retry_count*, *MAX_RETRIES*)] until a specified maximum number of retries is reached.

2.5. Summary

The current chapter first discusses about non-functional requirements that are essential for improving the quality of a system. Secondly, it presents Aspect-Oriented Programming (AOP), a maturing methodology that addresses non-functional requirements. An

AOP language, AspectJ (an extension of Java) is presented along with two examples. Thirdly, this chapter discusses how refactoring techniques enhance the structure of the code without changing its external functionality. Finally, the focus is on design patterns, which are used for solving recurring design issues following an already defined solution.

3. General Approach

Consider a system that requires additional non-functional features. The various options of addressing non-functional features are:

1. Refactor/apply without using patterns and aspects.
2. Refactor/apply using patterns.
3. Refactor/apply using aspects.
4. Refactor/apply using both patterns and aspects.

The above-mentioned options are represented diagrammatically in Figure 3.1.

A non-functional feature (security in the form of encryption) added to the case study (discussed in section 4) is implemented (experimented in section 5.4.1) in all the 4 approaches mentioned above. In practice, when a non-functional feature is addressed without using both patterns and aspects; it might take more time to understand as the familiarity that comes along with patterns is missing; with the absence of aspects, code pertaining to non-functional features would be scattered and intertwined along with the primary system code, which leads to less readability and understandability. If only patterns are used, they offer proven well-known solutions but the non-functional feature code will be scattered and intertwined along with primary system code. As the patterns are well known, a code written using them is easily understandable by other people. On the other hand, if only aspects are used, the non-functional feature code will be separated and not scattered but the solution is not widely known as that of patterns. The present research emphasize

on the fourth option that is to refactor/apply using both patterns and aspects. As both patterns and aspects offer a good design there is no need to trade-off one for another.

It would be worthwhile to experiment whether the recommended approach could be extended to functional features as well. Aspects have been used previously to address functional features in [18].

Figure 3.1: Options for addressing non-functional features

This chapter analyzes four different ways of addressing non-functional features; the advantages and disadvantages of one solution over the other.

4. Case Study -- Instant Messenger Application

As a case study, an existing open-source instant messenger system namely, Universal Instant Messenger (UIM) [8] is considered. It follows (distributed) client-server architecture. The relationship between its single server and multiple clients is illustrated in the Figure 4.1. Overlapping rectangles are used in the figure to represent multiple clients.

Figure 4.1: Architecture of UIM

4.1. Architecture of UIM

The architecture of the UIM system is shown in Figure 4.1. The control flows from *ClientGui* to *Client/ClientExtn* and from there to *Server*. This chapter discusses two versions

of the UIM⁴, one before (call it original) and another (call it enhanced) after enhancements. The enhancements are non-functional features applied using the proposed approach (implementation using both patterns and aspects) for improving the original UIM. Basic functionalities (described as scenarios in section 4.2) that are required for a typical instant messenger application are present in the UIM.

4.2. UIM Scenarios

During the case study, it has been identified that the UIM has the following eight scenarios. The implementation details are discussed in detail in the following sections.

4.2.1. *Registration*

The registration process allows to create a new user account i.e., allows to register an user with a new user Id and password combination, to the messenger system. This scenario ensures that the userId is unique for each user.

4.2.2. *Cancellation*

Cancellation process allows registered users to delete their account (unregister) from the messenger system.

⁴ In this chapter, general use of UIM implies both the original and enhanced versions.

4.2.3. *Login*

Login process allows registered users to gain access to their accounts. The entered userId and password are authenticated with that of the system's data and upon success, user(s) gain access to their own account. After gaining the access, operations including add a friend, remove an existing friend, converse with a friend, and change one's own status could be performed.

4.2.4. *Logout*

Logout process allows a logged-in user to logoff from UIM. When user(s) logoff, their active list of friends (those have been registered to the system) are automatically notified, about their unavailability.

4.2.5. *Add Friend*

Add friend process allows logged-in user(s) to add other users (those have been registered to the system) as their friends.

4.2.6. *Remove Friend*

The remove friend process allows logged-in users to remove existing friends from their list of buddies.

4.2.7. *Send Message*

The send message process allows logged-in users to send message to their friends (already in the list of buddies and whose status is available).

4.2.8. *Change Status*

The change-status process allows logged-in users to change their status to one of the following states: AVAILABLE, BUSY, INVISIBLE, or LOGOUT.

4.3. Design of UIM before enhancement

This section describes the relationship between classes; deficiencies in non-functional features in the original UIM before applying the proposed solution.

4.3.1. *Classes and their Roles*

The original design of UIM consists of the following classes:

1. Business classes: *Agent*⁵, *Human Agent*⁶ (sub class of *Agent*), *AgentInfo*, *Server* and *UimRegistry*.
2. User interface (UI) classes: The following four⁷ classes, *AgentGui*, *FriendPanel*, *LifeCyclePanel* and *StatusPanel* constitute the user interface of UIM.

⁵ Agent class represents Client in Figure 4.1

⁶ HumanAgent class represents ClientExtn in Figure 4.1

⁷ UI classes represents ClientGui in Figure 4.1.

3. Event and Listener classes: *InstantMessageEvent*, *InstantMessageListener*, *StatusChangeEvent* and *StatusChangeListener*.
4. Three exception-handling classes.

Class diagram of the original UIM (before enhancement) is presented in Figure 4.2

4.3.2. Feature Deficiencies

The following are some interesting non-functional features in the original UIM that needs enhancements:

4.3.2.1. Deficiency in Maintainability

In the original version of UIM, the server and the client components are distributed using Remote Method Invocation (RMI) [4]. The RMI code for this middleware distribution layer is coupled with the business classes. This tight coupling would be hard to maintain in the long run, if the middleware distribution layer needs to be replaced with a different technology. The maintainability of the system could be improved by decoupling the middleware layer from the business classes.

4.3.2.2. Deficiency in Serviceability

In the original version of UIM, all client (user) profiles including the added list of buddies would be lost, when the server shuts down. In Java, the server shut down implies the shutdown of Java virtual machine (JVM). Therefore, the registered user's profile could exist only as long as the JVM remains running. This implies that after the JVM is restarted, the user has to once again register. For a program that requires user authentication, it is essential to preserve the registered users' profiles beyond a single lifetime of the virtual machine. Persistence is the only way to achieve this.

4.3.2.3. Deficiency in Security

In the original version of UIM, the user Id and password are sent as clear text in a public network. A malicious attacker can easily get hold of the user Id and password data and cause damages. The damages could eventually affect the owners (users) as well as their buddies. The malicious attacker may pretend as an owner and send viruses/faulty messages to the buddies, or may even cause a denial of service to the owner by changing the password. Encrypting the user Id and password could eliminate this security breach.

4.3.2.4. Deficiency in Scalability & Agility

In the original version of UIM, all communications between server and clients are performed synchronously. In this synchronous communication, a client placing a request to the server has to wait to place the next request until the server completes processing the former request. If that request happens to be low in priority and/or require very long time to process, it would obviously take more time for the server to look into the next client request, leaving the client to wait unreasonably. Overall, this increases the response time (decreases the agility and the scalability of the system). To enhance the response time, scalability and agility, the low priority and slow synchronous processes needs to be converted into asynchronous multi-threaded processes.

4.3.2.5. Deficiency in Durability

In the original version of UIM, when a client tries to communicate with the server and if the server is temporarily unavailable or busy due to network problems, the client gives up immediately after the first attempt. Network problems like power shutdown and unavailability of resources are expected to happen in some critical circumstances. Ideally it is better to allow the clients to retry communication automatically avoiding an abrupt disconnection of the client from the server in the middle (or initial phase) of communication. Automatic retry would enable the system to be more durable.

Figure 4.2 Class Diagram of UIM before enhancement

4.4. Design of UIM after enhancement

The interesting features discussed in previous section 4.3.2 are enhanced using the proposed solution as follows:

4.4.1. Maintainability Enhancement

From section 4.3.2.1, it is identified that the middleware distribution layer needs to be decoupled from the business code to ease the system maintenance. In the enhanced UIM, this maintenance feature is added as aspects (collobarator mediator aspects described in section 5.1) using the Collobarator Mediator pattern principle.

4.4.2. Serviceability Enhancement

From section 4.3.2.2, it is identified that persistence is needed to enhance the system. Object serialization is the mechanism that is chosen, designed and implemented for the UIM. In order to enhance the serviceability of the application, persistence is added as aspects (decorator wrapper aspects described in section 5.2) applying the Decorator Wrapper pattern to store client profiles permanently.

4.4.3. *Accountability Enhancement*

From section 4.4.2, the serviceability of the system is enhanced with the persistence of client profile. This enhancement leads to a new enhancement, accountability, i.e. multiple clients persisting (writing to a shared resource) their profiles simultaneously, does not corrupt data of others. In the UIM, data corruption is highly possible, as several concurrent clients would access the shared code responsible for persistence. This persistence code is a critical section and needs to be accessed only by one client at a given time. In order to enhance the accountability of the system, protection of critical sections of code is added as an aspect (a code locking aspect described in section 5.3) using the Code Locking pattern.

4.4.4. *Security Enhancement*

From section 4.3.2.3, it is identified that encryption is needed to enhance the security of the system. In the original version of the UIM, the user Id and password are stored only in hashtables until the JVM of the server shuts down and are sent as clear text in the public network. But with the enhanced version, the user Id and password (confidential) data are stored both in hashtables and persisted in files as clear text and are sent as the same clear text in the public network. If the confidential data is sent or stored in clear text it is susceptible to malicious attacks. In order to enhance the security of the application, encryption is added as an aspect (a state aspect described in section 5.4) using the State pattern principle.

4.4.5. Scalability Enhancement

From section 4.3.2.4, it is identified that multiple threads are needed to perform asynchronous communications to enhance the scalability of the system. In the enhanced UIM, this scalability feature is added as an aspect (a thread pool aspect described in section 5.5) to broadcast of client's status update using the Thread Pool pattern principle.

4.4.6. Durability Enhancement

From section 4.3.2.5, it is identified that automatic retry is needed to enhance the durability of the system. In order to enhance the durability of the application, automated multiple retries is added as an aspect (a delayed retry aspect described in section 5.6) using the Delayed Retry pattern principle.

4.5. Class Diagram of UIM After Enhancement

The original UIM refactored using the proposed approach is shown in Figure 4.3.

4.6. Enhanced User Interface of UIM

The original user interface of UIM is enhanced with the addition of image icons for each button component. The enhanced UIM user interface is shown in Figure 4.4.

Figure 4.3 Class Diagram of UIM after enhancement

Figure 4.4: User interface of UIM

4.7. Summary

This chapter evaluates the proposed approach using a case study (Universal Instant Messenger), an instant messenger application. As part of the evaluation, five new non-functional features (serviceability, accountability, security, scalability and durability) are added and one non-functional feature (maintainability) is refactored. Design of UIM before and after enhancements is described in detail using class diagrams, snapshots, etc.

5. Implementation

The GoF patterns applied in this research belong to Behavioral and Structural categories. It should be noted that the GoF patterns discussed and implemented in this research are different from GoF Design Patterns in Java and AspectJ illustrated in [3], [20]. In there, a basic example is considered for implementing all patterns belonging to behavioral, structural and creational categories. Whereas, this research focuses on improving a real-world system (UIM) and the patterns required for the improvements of non-functional features alone are considered and implemented using the proposed approach (implementation using both patterns and aspects). In addition, identified concurrency patterns are applied to solve issues in the concurrent UIM. These patterns are implemented as aspects to address/refactor identified non-functional features within the UIM.

The Behavioral, Structural and Concurrency aspects are added to enhance the UIM [8]. The original version of UIM was developed in Java using only OOP principles. It employs Remote Method Invocation (RMI) [4] for a distributed environment and local method calls for a local (single machine) environment. The present research puts an emphasis on the distributed environment. The focus of the original UIM has been on the functional requirements of instant messenger. However, concurrency requirements have not been addressed in the original design. The same observations about many other applications could be made, based on systems that have a middleware layer for distribution. The use of such middleware introduces many multi-threading issues that developers of these systems are often unaware of. It should be noted that UIM is only used as a con-

crete vehicle for our research and our results do not depend on its use. The following aspects are implemented using AspectJ [1]. For the purpose of demonstration, the Code Locking and Delayed Retry patterns are presented both as abstract and concrete standalone aspects. Collaborator Mediator, Decorator Wrapper, State and Thread Pool patterns on their parts are illustrated as concrete standalone aspects.

5.1. Collaborator Mediator Aspect

The Collaborator Mediator pattern [15], is a behavioral pattern proposed in the GoF patterns. It is derived from the case study (section 4) for refactoring the coupled collaborator objects to prevent tight coupling. This pattern decouples two or more objects by introducing a mediator, which is demonstrated in Figure 5.1.

Figure 5.1: Collaborators and a Mediator

The decoupled objects, then communicate with each other only through the newly introduced mediator.

Table 5.1, illustrates the Collaborator Mediator aspect.

Table 5.1: Collaborator Mediator Aspect-1

```
public aspect RmiServerMediatorAspect {

    declare parents: Server extends UnicastRemoteObject;
    declare parents: Server implements RMIserverInterface;

    private RemoteUimRegistryInterface Server. RemoteRegistryInterface;

    public Server.new() throws Exception{
        Naming.bind ("uim21.server", this);
        UimRegistry r = new UimRegistry();
        regis = (RemoteUimRegistryInterface) Naming.lookup("rmi://localhost/uim21.registry");
    }

    public pointcut changeAgentStatusInTheirBuddyList
    (String agentId, String password, int status)
        : execution(public void Server.changeStatus(String, String, int)) && args(agentId,
        password, status);

    void around(String agentId, String password, int status)
        throws InvalidPasswordException, NoSuchAgentException :
        changeAgentStatusInTheirBuddyList(agentId, password, status)
        {
            proceed(agentId, password, status);
            AgentInfo ai = Server.getRealKey(agentId);
            Hashtable buddyTableCopy = (Hashtable)Server.buddyTable.clone();
            Hashtable clientToCallBackCopy = (Hashtable)UimRegistry.registry.clone();
            Iterator it = buddyTableCopy.keySet().iterator();
            while (it.hasNext()) {
                AgentInfo key = (AgentInfo) it.next();
                if ( ((Set) buddyTableCopy.get(key)).contains(ai) && !(key.getStatus()==Agent.LOGGEDOFF)) {
                    try {
                        CallbackClientInterface a = (CallbackClientInterface)
                            clientToCallBackCopy.get(key.getAgentId());

                        if (a != null) { // a can be null if I'm my own friend
                            a.handleStatusChange(new StatusChangeEvent(agentId, status) );
                        } //End of if
                    } //End of try
                    catch (RemoteException e) {System.out.println("remote exception"); }
                    } //End of if
            } //End of while
            //Update status only now to avoid sending updates to myself (if I'm my own friend) before I'm ready
            ai.setStatus(status);
        }

    public void Server.shutDown() {
        Naming.unbind("uim21.server");
    }

    private static aspect SofteningRemoteException{
        declare soft: java.rmi.RemoteException
        : (call (* Server.*(..))
            && !call(* Server.getRealKey(..))
            && !call(* Server.passwordValid(..));
        }
    }
}
```

For the case study, the middleware RMI layer already implemented, using the OO methodology is refactored using the AO methodology and collaborator mediator design pattern. As the middleware, is RMI [5] there are certain requirements that need to be adopted. They are listed as follows:

1. An interface that extends an interface - `java.rmi.Remote` - should be defined. The required methods for this newly created interface are defined to access from any JVM. As the methods in an interface do not have any concrete implementation, these methods are implemented by an object implementing the interface.
2. The concrete implementation of the newly defined interface should:
 - a. Extend the defined interface.
 - b. Define a constructor that throws an Exception. An Exception is mandatory as there might be some remote exceptions.
 - c. Provide concrete implementation for each remote method in the defined interface.
3. The concrete object that implements the defined interface should extend the `java.rmi.server.UnicastRemoteObject`

The Server and Agent are the two main classes of the UIM. The RMI version of the UIM implemented in OO methodology for the Server class is provided in Appendix - B.1. Highlighted sections of OO implementation provided in Appendix - B.1, shows that the implementation becomes tightly coupled with the RMI code. In OOP, if the middleware layer is changed from RMI to a different one (say, CORBA) then the implementation needs to be modified accordingly.

The proposed solution in this research combines aspect-oriented programming, refactoring techniques and design patterns (Collaborator Mediator pattern) to simplify the middleware implementation layer. The result of this combination is the Collaborator Mediator aspect. It demonstrates how the tight couplings of collaborators (according to the Collaborator Mediator pattern, the Server and its clients are referred to collaborators) in the object-oriented implementation are decoupled. The Collaborator Mediator Aspect acts as a mediator between its collaborators.

The Collaborator Mediator Aspect's implementation details are as follows:

The aspect follows the requirements needed for RMI mentioned above in this section.

According to the requirements, this aspect makes the Server class extend the `java.rmi.server.UnicastRemoteObject` and implement a new interface - `RMIServerInterface` that extends the `java.rmi.Remote` interface.

A constructor that throws an Exception is defined for the Server object using `Server.new`. Inside this constructor, the remote Server registers its own object (using *this*), bound to RMI registry (a network-based RMI registry to keep track of the distributed objects) with a unique name (`uimServer` is the unique name for this Server). Remote client objects need to know the server's name in the RMI registry to communicate.

The RMIServerInterface has a method named *changeStatus* that needs to be implemented in the Server. The pointcut *changeAgentStatusInTheirBuddyList* captures call to the *changeStatus* method. RMI code required to change remotely the status of appropriate Agent(s) is placed in the around advice of this pointcut.

Other than the RMI registry, the server side stores a registry – UimRegistry, by storing the clients (called as Agent objects) that are logged on to the Server. This UimRegistry is created in the constructor of the Server (UimRegistry r = new UimRegistry()). This UimRegistry implements an interface – RemoteUimRegistryInterface, which in turn extends the java.rmi.Remote interface to be accessed remotely by clients (Agent). A variable (field) *RemoteRegistryInterface* of type RemoteUimRegistryInterface is defined to store remote Agents that are logged on to the server. As the system is based on a single server and multiple clients' architecture, there could be more than one Agent accessing the Server at a given time. The references to the Agents are stored in the server side for enabling the Server to contact the appropriate Agent(s) when required. According to the requirements of RMI, every remote method needs to either throw a RemoteException or to be surrounded by a try/catch block to handle the RemoteException. There is an inner aspect - SofteningRemoteException that efficiently avoids the need for this try/catch block (or throws clause) when compared to the conventional (OO) implementation (listed in Appendix-B.1). In the Appendix-B.1, the *throws RemoteException* of the RMI code is scattered into almost every method. In the AO approach the inner aspect - SofteningRemoteException - in Table 5.1 has replaced the interdependence of RMI code and extracted the *throws RemoteException* into one separate aspect.

The decoupling concept applied to the server side is now applied to the client side as well. Appendix - B.2, demonstrates the implementation details of the client side in AO approach.

Using the collaborator mediator aspect, the maintainability of the system is improved by changing from one middleware software to another. This requires changes only to the mediator aspects; the primary code is untouched.

5.2. Decorator Wrapper Aspect

The Decorator Wrapper Pattern [15] is a structural pattern proposed in the GoF patterns. It is derived from the case study (section 4) to persist serialized objects into a file. The Decorator Wrapper Pattern suggests dynamically wrapping one wrapper inside another to enhance or withdraw functionalities. For example, GUI component - scroll-pane - is a decorator that can be used in any of the following GUI components: text area, window and panel.

```

public aspect UimPersistenceAspect {

    pointcut writeStorage() :
        execution(public void Server.subscribe(..))
        || execution(public void Server.unsubscribe(..))
        || execution(public AgentInfo Server.addFriend(..))
        || execution(public AgentInfo Server.removeFriend(..)) ;

    after():writeStorage(){
        FileOutputStream fos1 = null;
        FileOutputStream fos2 = null;

        try{
            fos1 = new FileOutputStream("ServerAgentTable.ser");
            ObjectOutputStream oos1 = new ObjectOutputStream(fos1);

            oos1.writeObject(Server.buddyTable);
            fos2 = new FileOutputStream("ServerPwdTable.ser");
            ObjectOutputStream oos2 = new ObjectOutputStream(fos2);
            oos2.writeObject(Server.passwordTable);
        }
        catch (FileNotFoundException e){}
        catch (IOException e1) {}
    }
}

```

Figure 5.2: Decorator Wrapper Aspect For Writing

Figure 5.2, illustrates a concrete decorator wrapper aspect. Server class in Appendix-B.1 illustrates the primary class to which the advice in Figure 5.2 is applied. The decorator wrapper aspect's implementation details are as follows. In the UIM case study, user (Agent) profiles are stored in hashtables (for buddyTable – user object is the key and list of user's buddies is the value and for passwordTable – userId is the key and password is the value). The pointcut, *writeStorage* captures execution of methods *subscribe*, *unsubscribe*, *addFriend* and *removeFriend* in the Server. Because the user profiles, stored temporarily in hashtables (temporary because, if the JVM shuts down the hashtables will loose the data they are holding), tend to change after the execution of the above four methods the *writeStorage* pointcut captures them. Therefore, in the after advice of the

pointcut *writeStorage* (i.e., after executing anyone of the above 4 methods), the hashtables (buddyTable and passwordTable) use an ObjectOutputStream to serialize themselves into files (ServerAgentTable.ser and ServerPwdTable.ser respectively) for permanent storage on the server side. The extension ‘ser’ on these file names represent their serializable nature.

```
public aspect UimReadPersistenceAspect {
    Hashtable tempAgentTable = new Hashtable();
    Hashtable tempPasswordTable = new Hashtable();

    pointcut readStorage()
        : call(Server.new());

    after() : readStorage() {
        FileInputStream fis1 = null;
        FileInputStream fis2 = null;
        try{
            fis1 = new FileInputStream("ServerAgentTable.ser");
            ObjectInputStream ois1 = new ObjectInputStream(fis1);
            tempAgentTable = (Hashtable)ois1.readObject();
            Server.buddyTable = tempAgentTable;
            Iterator it = Server.buddyTable.keySet().iterator();
            while (it.hasNext()) {
                AgentInfo key = (AgentInfo) it.next();
                if ( Server.buddyTable.get(key) != null){
                    key.setStatus(Agent.LOGGEDOFF);
                }
            }
        }
        fis2 = new FileInputStream("ServerPwdTable.ser");
        ObjectInputStream ois2 = new ObjectInputStream(fis2);
        tempPasswordTable = (Hashtable)ois2.readObject();
        Server.passwordTable = tempPasswordTable;
    }
    catch (FileNotFoundException e){}
    catch (IOException e1) {}
    catch (ClassNotFoundException e2) {}
}
```

Figure 5.3: Decorator Wrapper Aspect For Reading

Using the Decorator Wrapper Pattern, the `ObjectOutputStream` decorates the `FileOutputStream`, dynamically adding a new capability to be able to serialize an Object into a file. The `ObjectOutputStream` is a decorator that is used commonly to enhance the functionality of concrete components like `FileOutputStream` and `ByteArrayOutputStream`s.

Figure 5.3, illustrates another concrete decorator aspect to read from the serialized file. The pointcut, *readStorage* captures the execution of the Server's constructor. This aspect comes into play on rare occasions when the server needs to be restarted. Therefore, in the after advice of the pointcut *readStorage* (i.e., immediately, after rebooting the server) `ObjectInputStream` decorates the `FileInputStream` dynamically adding a new capability to be able to reload objects (in this case, the objects are hashtables) from serialized files.

Using the Decorator Wrapper aspect, the serviceability of the system is improved since objects are available even after the reboot of JVM.

5.3. Code Locking Aspect

The Code Locking pattern suggests locking of the critical section and allows only one thread to process the locked code at a given time. After completing the process, the lock is released so that another thread that needs to process the critical section could acquire the exclusive lock and do its processing. Locks are issued on a first-come first-serve basis. This pattern also suggests that the use of locks should be minimized to keep the system speed high.

```

public abstract aspect CodeLockingAbstractAspect {
    private Lock commonResource = new <lock()>;
    public abstract pointcut criticalSection();
    before() : criticalSection() {
        commonResource.lock();
    }
    after() : criticalSection() {
        commonResource.unlock();
    }
}

```

Figure 5.4: Abstract Code Locking Aspect

```

public aspect CodeLockingConcreteAspect extends
    CodeLockingAbstractAspect {

    public pointcut criticalSection () :
        execution(public void Server.subscribe(..))
        || execution(public void Server.unsubscribe(..))
        || execution(public AgentInfo Server.addFriend(..))
        || execution(public AgentInfo Server.removeFriend(..));
}

```

Figure 5.5: Concrete Code Locking Aspect

Figure 5.4, illustrates a standalone abstract code locking aspect. This aspect would not be required without the decorator wrapper aspect discussed in section 5.2. The implementations of the Code Locking Aspect details are as follows. A variable (field) *commonResource* of type Lock is defined and this creates a new Lock object. There are various APIs available for code locking. An abstract pointcut *criticalSection* is defined to be implemented later in a concrete aspect that extends the CodeLockingAbstractAspect. In the concrete aspect that extends the CodeLockingAbstractAspect, the concrete *criticalSection* pointcut captures execution to all critical sections of the code. The before() advice instructs the captured critical section to lock before entering the critical section. The after()

advice instructs the captured critical section to unlock after exiting the critical section. Thus, the critical section is protected without modifying the primary system.

In the UIM application, as an after-thought clients' information in the server is persisted by means of aspects (discussed in section 5.2) to enhance serviceability. Only a single thread reads the persisted information. Therefore, the reading portion of the persistence code is not a critical section. On the other hand, several concurrent threads could write to the persisted information. Therefore, the writing portion of the persistence code is considered as a critical section. As multiple concurrent clients could access the common persistence storage at the same time, there is a need of protecting this critical section to prevent data corruption. Therefore, the code locking aspect illustrated in Figure 5.5 is applied at critical sections. Appendix - B.1, illustrates the UIM's Server class, which the advice in Figure 5.5 is applied. In addition, it should be noted that the critical section written by the Decorator Wrapper aspect would not be visible in the primary code (the Server class) as the aspect code is isolated from the primary functional code.

Locking and concurrency control go hand in hand. Therefore, the Code Locking abstract aspect designed using the Code Locking pattern in Figure 5.4 could be applied whenever there is a need to prevent a critical section in a concurrent system.

Using the Code Locking aspect, the accountability of the system is improved, as the data corruption does not happen.

5.4. State Aspect

The State pattern [15] is a behavioral pattern proposed in the GoF patterns. It is applied to changes in the state of objects from one to another. It is derived from the case study (section 4) for encrypting confidential data (userId and password) with one of the two options (states), a 64-bit key or a 128-bit key.

```
public aspect UimEncryptAspect {

    declare error :
        (call(public void LifeCyclePanel.setAgentId(String))
         | | call(public void LifeCyclePanel.setPassword(String)))
        && !within(UimEncryptAspect)
        && !within(LifeCyclePanel)
        : "SetAgentId and SetPassword are not accessible to you";

    AbstractEncrypter encrypter;
    If bit64 == true { // bit64 is a state variable that decides one of the two encryption options
        encrypter = new PBEEncrypter("");
    }
    else{
        encrypter = new PBEEncrypter128("");
    }

    pointcut encrypt(LifeCyclePanel lcg, String agentId):
        call(public void LifeCyclePanel.setAgentId(String))&& args(agentId)&&
        withincode(* LifeCyclePanel.*(..)) &&
        !withincode(* LifeCyclePanel.setAgentId(..))
        && target (lcg);

    void around(LifeCyclePanel lcg, String agentId): encrypt(lcg, agentId){
        String agentID = null;
        agentID = encrypter.encrypt(agentId);
        lcg.setAgentId(agentID);
    }

    pointcut encryptPassword(LifeCyclePanel lcg, String password):
        call(public void LifeCyclePanel.setPassword(String))&& args(password)
        && withincode(* LifeCyclePanel.*(..)) &&
        !withincode(* LifeCyclePanel.setPassword(..))
        && target (lcg);

    void around(LifeCyclePanel lcg, String password): encryptPassword(lcg, password){
        String pwd = null;
        pwd = encrypter.encrypt(password);
        lcg.setPassword(pwd);
    }
}
```

Figure 5.6: State Aspect


```

import java.io.UnsupportedEncodingException;
import javax.crypto.*;
import javax.crypto.spec.*;
public class PBEencrypter {
    Cipher ecipher, dcipher;
    // 8-byte Salt
    byte[] salt = { (byte)0xA9, (byte)0x9B, (byte)0xC8, (byte)0x32,
                    (byte)0x56, (byte)0x35, (byte)0xE3, (byte)0x03 };
    // Iteration count
    int iterationCount = 19;

    public PBEencrypter(String passPhrase) {
        try {
            // Create the key
            PBEKeySpec keySpec = new PBEKeySpec(passPhrase.toCharArray(), salt,
            iterationCount);
            SecretKey key = SecretKeyFactory.getInstance(
                "PBKDF2WithHmacSHA1").generateSecret(keySpec);
            ecipher = Cipher.getInstance(key.getAlgorithm());
            dcipher = Cipher.getInstance(key.getAlgorithm());

            // Prepare the parameter to the ciphers
            PBEParameterSpec paramSpec = new PBEParameterSpec(salt, iterationCount);

            // Create the ciphers
            ecipher.init(Cipher.ENCRYPT_MODE, key, paramSpec);
            dcipher.init(Cipher.DECRYPT_MODE, key, paramSpec);
        }
        catch (java.security.InvalidAlgorithmParameterException e) {}
        catch (java.security.spec.InvalidKeySpecException e) {}
        catch (javax.crypto.NoSuchPaddingException e) {}
        catch (java.security.NoSuchAlgorithmException e) {}
        catch (java.security.InvalidKeyException e) {}
    }

    public String encrypt(String str) {
        try {
            // Encode the string into bytes using utf-8
            byte[] utf8 = str.getBytes("UTF8");
            // Encrypt
            byte[] enc = ecipher.doFinal(utf8);
            // Encode bytes to base64 to get a string
            return new sun.misc.BASE64Encoder().encode(enc);
        }
        catch (javax.crypto.BadPaddingException e) {}
        catch (IllegalBlockSizeException e) {}
        catch (UnsupportedEncodingException e) {}
        return null;
    }
}

```

Figure 5.7: Class – PBEencrypter with 64-bit key

```

public void actionPerformed(ActionEvent e) {
    String agentName;
    String agentId;
    String password;
    String encryptedAgentId = null;
    String encryptedPassword = null;
    AbstractEncrypter encrypter;
    if bit64 == true { // bit64 is a state variable that decides one of the two encryption options
        encrypter = new PBEEncrypter("");
    }
    else{
        encrypter = new PBEEncrypter128("");
    }
    String label = e.getActionCommand();
    if (label.equals("Register")) {
        agentName = JOptionPane.showInputDialog("Enter your Name");
        agentId = JOptionPane.showInputDialog("Enter new User Id");
        password = JOptionPane.showInputDialog("Enter a new password");
        encryptedAgentId = encrypter.encrypt(agentId);
        encryptedPassword = encrypter.encrypt(password);
        .
        .
        .
        boolean serverAvailable = HumanAgent.register(agentName, encryptedAgentId,
        encryptedPassword);
        .
        .
        .
    }
    else if (label.equals("Login")) {
        agentId = JOptionPane.showInputDialog("Enter Your User Id");
        password = JOptionPane.showInputDialog("Enter Your password ");
        encryptedAgentId = encrypter.encrypt(agentId);
        encryptedPassword = encrypter.encrypt(password);
        .
        .
        .
        HumanAgent myAgent = (HumanAgent) HumanAgent.login(encryptedAgentId,
        encryptedPassword);
        .
        .
        .
    }
    else if (label.equals("UnRegister")) {
        agentId = JOptionPane.showInputDialog("Enter Your User Id");
        password = JOptionPane.showInputDialog("Enter Your password ");
        encryptedAgentId = encrypter.encrypt(agentId);
        encryptedPassword = encrypter.encrypt(password);
        .
        .
        .
        HumanAgent.unregister (encryptedAgentId, encryptedPassword);
        .
        .
        .
    }
}
}

```

Figure 5.8: Approach without using patterns and aspects

```

public void actionPerformed(ActionEvent e) {
    String agentName;
    String agentId;
    String password;
    String encryptedAgentId = null;
    String encryptedPassword = null;
    AbstractEncrypter encrypter;
    If bit64 == true { // bit64 is a state variable that decides one of the two encryption options
        encrypter = new PBEEncrypter("");
    }
    else{
        encrypter = new PBEEncrypter(128(""));
    }
    String label = e.getActionCommand();
    if (label.equals("Register")) {
        agentName = JOptionPane.showInputDialog("Enter your Name");
        agentId = JOptionPane.showInputDialog("Enter new User Id");
        password = JOptionPane.showInputDialog("Enter a new password");
        encryptedAgentId = encrypter.encrypt(agentId);
        setAgentId(encryptedAgentId);
        encryptedPassword = encrypter.encrypt(password);
        setPassword(encryptedPassword);
        .
        .
        .
        boolean serverAvailable = HumanAgent.register(agentName, encryptedAgentId,
        encryptedPassword);
        .
        .
        .
    }
    else if (label.equals("Login")) {
        agentId = JOptionPane.showInputDialog("Enter Your User Id");
        password = JOptionPane.showInputDialog("Enter Your password ");
        encryptedAgentId = encrypter.encrypt(agentId);
        setAgentId(encryptedAgentId);
        encryptedPassword = encrypter.encrypt(password);
        setPassword(encryptedPassword);
        .
        .
        .
        HumanAgent myAgent = (HumanAgent) HumanAgent.login(encryptedAgentId,
        encryptedPassword);
        .
        .
        .
    }
    else if (label.equals("UnRegister")) {
        agentId = JOptionPane.showInputDialog("Enter Your User Id");
        password = JOptionPane.showInputDialog("Enter Your password ");
        encryptedAgentId = encrypter.encrypt(agentId);
        setAgentId(encryptedAgentId);
        encryptedPassword = encrypter.encrypt(password);
        setPassword(encryptedPassword);
        .
        .
        .
    }
}

```

Figure 5.9: Approach using only patterns

```

public aspect UimEncryptAspect {
pointcut encrypt(LifeCyclePanel lcg, ActionEvent event):
    call(public void LifeCyclePanel. actionPerformed (ActionEvent))&& args(event);
void around(String agentId): encrypt(event){
    String agentName;
    String agentId;
    String password;
    String encryptedAgentId = null;
    String encryptedPassword = null;
    AbstractEncrypter encrypter;
    If bit64 == true // bit64 is a state variable that decides one of the two encryption options
        encrypter = new PBEEncrypter("");
    else
        encrypter = new PBEEncrypter128("");
    String label = event.getActionCommand();
    if (label.equals("Register")) {
        agentName = JOptionPane.showInputDialog("Enter your Name");
        agentId = JOptionPane.showInputDialog("Enter new User Id");
        password = JOptionPane.showInputDialog("Enter a new password");
        encryptedAgentId = encrypter.encrypt(agentId);
        encryptedPassword = encrypter.encrypt(password);
        .
        .
        .
        boolean serverAvailable = HumanAgent.register(agentName, encryptedAgentId,
        encryptedPassword);
        .
        .
        .
    }
    else if (label.equals("Login")) {
        agentId = JOptionPane.showInputDialog("Enter Your User Id");
        password = JOptionPane.showInputDialog("Enter Your password ");
        encryptedAgentId = encrypter.encrypt(agentId);
        encryptedPassword = encrypter.encrypt(password);
        .
        .
        .
        HumanAgent myAgent = (HumanAgent) HumanAgent.login(encryptedAgentId,
        encryptedPassword);
        .
        .
        .
    }
    else if (label.equals("UnRegister")) {
        agentId = JOptionPane.showInputDialog("Enter Your User Id");
        password = JOptionPane.showInputDialog("Enter Your password ");
        encryptedAgentId = encrypter.encrypt(agentId);
        encryptedPassword = encrypter.encrypt(password);
        .
        .
        .
        HumanAgent.unregister (encryptedAgentId, encryptedPassword);
        .
        .
        .
    }
}
}

```

Figure 5.10: Approach using only aspects

```

public void actionPerformed(ActionEvent e) {
    String agentName;
    String agentId;
    String password;
    String label = e.getActionCommand();
    if (label.equals("Register")) {
        agentName = JOptionPane.showInputDialog("Enter your Name");
        agentId = JOptionPane.showInputDialog("Enter new User Id");
        password = JOptionPane.showInputDialog("Enter a new password");
        setAgentId(agentId);
        setPassword(password);
        .
        .
        .
        boolean serverAvailable = HumanAgent.register(agentName, agentId, password);
        .
        .
        .
    }
    else if (label.equals("Login")) {
        agentId = JOptionPane.showInputDialog("Enter Your User Id");
        password = JOptionPane.showInputDialog("Enter Your password ");
        setAgentId(agentId);
        setPassword(password);
        .
        .
        .
        HumanAgent myAgent = (HumanAgent) HumanAgent.login(agentId, password);
        .
        .
        .
    }
    else if (label.equals("UnRegister")) {
        agentId = JOptionPane.showInputDialog("Enter Your User Id");
        password = JOptionPane.showInputDialog("Enter Your password ");
        setAgentId(agentId);
        setPassword(password);
        .
        .
        .
        HumanAgent.unregister (agentId, password);
        .
        .
        .
    }
}
}

```

Figure 5.11: Primary Code Captured by the patterns and aspects approach

Figure 5.6 illustrates a concrete implementation of the state aspect. Its implementation details are as follows. The pointcut *encryptUserId* captures a call that sets the agentId (userId) entered in the UIM's Graphical User Interface (GUI) [the GUI class responsible here is the LifeCyclePanel].

In the around advice, the `agentId` assigned initially in a plain text state is changed to an encrypted state using the `encrypt` method of this `PBEencrypter` class, listed in Figure 5.7. The `PBEencrypter` uses a `PBEWithMD5AndDES` algorithm to encrypt alphanumeric characters (also special characters like \$, are allowed). This algorithm is a symmetric encryption one using a same key for encryption as well as for decryption. It uses DES keys that are 64 bits in length. The `PBEencrypter128` class is similar to the `PBEencrypter` class except the former uses DES keys that are 128 bits in length. Typically, the encryption process brings in additional security at the expense of additional time needed to encrypt and decrypt. Therefore, to reduce this overhead time, symmetric encryption is chosen over asymmetric one because the former is faster than the latter. The pointcut *encryptPassword* encrypts password as the pointcut *encryptUserId* encrypts `userId`. Both these pointcuts have special keyword *withincode* and *!withincode*. The scope of the *withincode* in this aspect is to capture any joinpoint inside the lexical scope of all methods in the `LifeCyclePanel`. The scope of *!withincode* is to prevent stack-overflow, by calling itself. The *target* keyword captures joinpoints where the target object is `LifeCyclePanel`.

Sometimes it is sufficient that the password alone is encrypted. However, encrypting the `userId` gives additional security in preventing (or reducing) dictionary attacks. A malicious user could easily tap the `agentId` (in a clear text) when sent across the network and launch dictionary attacks on the unknown password. On the contrary, if both the `userId` and password are sent encrypted then a malicious attacker needs to crack two unknowns.

Finally, *declare error* is used to prevent any other object but the LifeCyclePanel and state aspect from calling the setAgentId and setPassword. This error declaration is performed to protect unintentional and/or accidental access to userId and password (as the setAgentId and setPassword methods are declared public).

The state aspect adds security without the knowledge of the UIM system's core code.

5.4.1. Comparison of the four general approaches to address security

The four general approaches discussed in section 3 are evaluated as below:

Implementation of the approach without using patterns and aspects is listed in appendix Figure 5.8. Highlighted sections of this implementation show that the non-functional feature (security) code is scattered and intertwined along with the primary system code, which leads to less readability and understandability.

Implementation of the approach using only patterns is listed in appendix Figure 5.9. Highlighted sections of this implementation shows that the non-functional feature (security) code is scattered and intertwined along with the primary system code and pattern code (shown in bold font), which again leads to less readability and understandability as the code becomes lengthy due to the additional non-functional feature.

Implementation of the approach using only aspects is listed in appendix Figure 5.10. Implementation of the proposed approach (using both patterns and aspects) is discussed in the beginning of section 5.4. When comparing the only-aspect approach with the proposed approach, in the former, the state pattern could not be applied. Therefore, *setAgentId()* method (for capturing an unencrypted state of *agentId*) and *setPassword()* method (for capturing an unencrypted state of *password*) in primary code (listed in Figure 5.11) could not be captured in the aspect; this makes the aspect code include even the primary code along with the appropriate security code. As the goal is to address the security feature in an efficient manner, it would be a good practice to exclude the primary code in aspects as illustrated in Appendix Figure 5.6 using the state pattern, to avoid unnecessary repetition of the primary code in both the primary class and the aspect. From the above four approaches, the patterns and aspects approach provides a better solution i.e. use of well-known state pattern and improves readability and understandability of both primary code and aspect code, by preventing code scatter and intertwines unlike the other approaches that do not use aspects. In addition, the length of the primary system code shrinks a lot due to the introduction of aspects. When scattered code is transformed into aspects, its code also shrinks. This contributes to the improvement of readability and understandability of a system.

5.5. Thread Pool Aspect

The Thread Pool pattern suggests, creating a bounded number of threads and reusing them for multiple tasks⁸. By reusing threads for more than one task, the thread crea-

⁸ A task is a request or an action that needs to be processed within a given system.

tion/deletion overhead is spread among the multiple tasks. When subsequent tasks (requests) arrive, the threads already exist and there is no overhead time for creating them. In addition, upon completion of an assigned task, overhead time for deleting the threads is eliminated. By using threads in a concurrent system, its response time could be reduced. Furthermore, by fine-tuning the number of threads in the thread pool, system resources could be managed in a controlled manner, thus preventing unexpected behaviors such as system crashes.

Figure 5.12: Sequence Diagram for Update Status Process as Conceived in the Original Version of UIM

Figure 5.12 illustrates sequential diagram for updating the status of a client among the clients listed in the associated buddy list, as conceived in the original UIM version. In Figure 5.12 when a client (user1) updates his/her status, the buddies of user1 are notified

about the status change synchronously. This means, the user1 needs to wait as long as the server notifies the change to all of his/her online buddies. For instance if user1 has a greater number of online buddies (say 500), he/she would have to wait for a considerable amount of time for his/her next request to be processed.

```
public aspect UimAsyncStatusThreadPoolAspect {
    public ClientUpdate cb;
    declare parents: Server implements ClientUpdate;

    public void Server.callBack(NotifyThread n,
        String arg1, String arg2, int arg3) {
        changeStatus(arg1, arg2, arg3);
    }

    //create pool of 3 threads
    public static AsyncClientCallBack Server.asyncClientCallback
        = new AsyncClientCallBack(3);

    pointcut initThreadPool()
        : call(Server.new());

    after() : initThreadPool() {
        Server.asyncClientCallback.start();
    }

    pointcut asyncChangeStatus(String agentId,
        String password, int status)
        : call(public void *.updateStatus(String, String, int)) &&
        args(agentId, password, status);

    void around(String agentId, String password, int status)
        : asyncChangeStatus(agentId, password, status){
        //Replaces the synchronous updateStatus
        //    with the AsyncClientCallBack
        //doClientUpdate method to
        //    asynchronously update the status.
    }
}
```

Figure 5.13: Thread Pool Aspect

To enhance the response time of the UIM application, synchronous update status process is converted to an asynchronous multi-threaded one by weaving thread pool aspect (illustrated in Figure 5.13) into the application. The aspect introduces a new interface - *ClientUpdate* (listed in Appendix-C.2), for the *Server* class. The only requirement for interfacing to the asynchronous thread pool is to implement the *ClientUpdate*. The *ClientUpdate* interface has one method, *callback()*. The *callback()* method has one parameter of type *NotifyThread*. The *NotifyThread* is a reference to the thread pool that is created and the remaining parameters are any arguments to pass to the *callback()* method (or null if one isn't specified).

Because, the *Server* implements *ClientUpdate*, the concrete *callback()* for the *Server* is implemented in the aspect. A new variable *asyncClientCallback* of type *AsyncClientCallback* is added and this creates a new *AsyncClientCallback* object and specifies a maximum number of threads required in the pool. The *AsyncClientCallback* class (listed in Appendix-C.1), encapsulates the configurable thread pool and provides the *doClientUpdate()* method to request an asynchronous update status. Also, note that the *AsyncClientCallback* class provides two forms of *doClientUpdate()*, one with and another without parameter passing capability. Depending upon the input parameters received one is used over the other. If the update requests are more in count compared to available threads (in pool) to perform the update status, the update requests are placed in a first in first out (FIFO) queue. When a thread completes its task and available for next one, the next request from the FIFO queue is retrieved and served. The pointcut *initThreadPool* captures joinpoint when the *Server* is created. The after advice for this pointcut instructs to start

the threads in the pool. The pointcut *asyncChangeStatus* captures call to the synchronous *updateStatus* method and in the around advice of this pointcut, the synchronous *updateStatus* is replaced with the asynchronous *updateStatus*, which is the *changeStatus* method.

Figure 5.14 shows the sequence diagram reflecting modifications, made by weaving *ThreadPoolAspect*, to the update status scenario. Apart from the classes already talked, additional classes referred in Figure 5.14, are *RequestClientAsyncCallBack* (listed in Appendix-C.3) and *NotifyThread* (listed in Appendix-C.4). If aspects have not been used, converting from synchronous to asynchronous communication would require deleting/adding code in the *Server* class. However, in the adopted approach the primary system code is left untouched.

Using the Thread Pool aspect, the scalability of the system is improved as new threads created respond to more number of user requests than before.

Figure 5.14: Modified Sequence Diagram for Update Status Process in the Enhanced Version of UIM

5.6. Delayed Retry Aspect

The Delayed Retry pattern suggests if a requested resource is not readily available for some reasons (e.g., network failure, resource request contention, resource limits), repeated requests is made in an intelligent way. Several common-channel allocation protocols [10], [24], [25], [28], [30], address the issue discussed in this pattern. The pattern provides an optimal solution in the place of giving up after the first time and relentlessly retrying, both of which can degrade system performance. It is especially essential for a distributed system, intended to have huge number of concurrent processes.

Figure 5.15 illustrates the abstract aspect code for Delayed Retry aspect. The initial values of two of the retry parameters—`MAX_RETRIES` and `INITIAL_DELAY`—are customizable according to the requirements of applied system. The pointcut for the aspect is defined as abstract. Concrete definition for the pointcut depends on the application system, and typically captures access request across distributed resources. When pointcuts are captured, they are replaced by around advice code. Using *proceed()* keyword, the around() advice (in Figure 5.15) instructs the captured pointcut (*retryWhenNotAccessible*) to execute its normal execution. If resource request attempts fail due to anticipated exceptions listed in the catch block of the *retryWhenNotAccessible* pointcut, repeated attempts for the resource are made until specified `MAX_RETRIES` is reached. In between the retries *sleepBetweenRetries* method is invoked. The *sleepBetweenRetries* puts the processes to sleep for exponentially increasing random period in function of *current_Retry_Count*.

The repeated attempts end if requested resource is successfully obtained or the MAX_RETRIES is reached.

```
public abstract aspect DelayedRetryAbstractAspect {
    double current_Retry_Count = 0;
    public final static double MAX_RETRIES = <Constant>;
    private final static int INITIAL_DELAY = <Constant>;

    public void sleepBetweenRetries(){
        // For brevity the method is written in pseudo-code.
        // sleep_Time = random(INITIAL_DELAY,
            current_Retry_Count, MAX_RETRIES);
        // sleep up to sleep_time
    } // Eof retry()

    public abstract pointcut retryWhenNotAccessible();

    Object around() :
        retryWhenNotAccessible (userName, userId, password){
        while(true){
            try{ return proceed(); }
            catch (Exception e){
                sleepBetweenRetries();
                if(++current_Retry_Count>=MAX_RETRIES) {
                    System.out.println("Have to give up retry! Please try again later");
                    System.exit(0);
                } //Eof if
            } //Eof Catch
        } //Eof While
    } //Eof Object around
} // Eof aspect
```

Figure 5.15: Abstract Delayed Retry Aspect

In UIM, a system running as the server is expected to be available all the time for communicating with clients. However, a client communication request could timeout if the communication network is experiencing disruptions or if the server is not available.

Therefore, it becomes necessary to intelligently retry connecting to the server under such unexpected circumstances to improve system durability.

```
public aspect DelayedRetryConcreteAspect extends
    DelayedRetryAbstractAspect {

    public pointcut retryWhenNotAccessible
        (String agentName, String agentId, String password)
        : call(public boolean *.register(String, String, String)) &&
            args(agentName, agentId, password);

} // Eof aspect
```

Figure 5.16: Concrete Aspect Inheriting Delayed Retry Aspect

There are multiple scenarios in UIM in which a client initiates communication with the server. One such scenario is when a new user attempts to register. Figure 5.17, illustrates the registration communication initiation code segment of UIM client. The concrete pointcut illustrated in Figure 5.16 corresponds to this method call. It should be noted that there are no marks in the original code segment, illustrated in Figure 5.17, indicating that an aspect is woven over it. Delayed Retry pattern is typically required in a system where multiple processes compete for shared resources. The adopted approach suggests that the Delayed Retry Aspect illustrated in Figure 5.15, could be readily plugged into any such system that requires automated retries without polluting the original code of the system.

Using the Delayed Retry aspect, the durability of the system is improved even beyond contention collisions such as network failures.


```

import java.rmi.*;

public static boolean register
(String agentName, String agentId, String password) throws
    RemoteException, ConnectException, NotBoundException {
    boolean serverAvailabe = false;
    if (rmiServer == null) {
        rmiServer = (RMIServerInterface) Naming.lookup("rmi://" +
            serverAddress + "/uimServer");
    }
    rmiServer.subscribe(agentName, agentId, password);

    registry = (RMIServerInterface) Naming.lookup("rmi://" +
        serverAddress + "/uimRegistry");

    serverAvailabe = true;
    return serverAvailabe;
} //Eof method register

```

Figure 5.17: Register method captured by Delayed Retry Aspect

5.7. Integration of Aspects

Integrating all the independent aspects discussed in sections from 5.1 to 5.6 needs to take care of priorities into account. For example, the code locking aspect (section 5.3) needs to be executed prior to the decorator wrapper aspect (section 5.2), as the intention of the former is to protect the latter. The keyword *declare precedence* as shown in Figure 5.18, illustrates this behavior.

```

public aspect SetPrecedenceForAspects {
    declare precedence:
        CodeLockingAbstractAspect, CodeLockingConcreteAspect, UimPersistenceAspect;
}

```

Figure 5.18: Set Precedence For Aspects

5.8. Summary

The design of UIM with enhancements (of non-functional features, maintainability, serviceability, accountability, security, scalability and durability) is implemented using pattern-based aspect approach. For the security feature, different ways of implementation discussed in chapter 3 are evaluated in detail.

6. Evaluation

The present research suggests that performance *ilities* of distributed applications (e.g., maintainability, serviceability, accountability, security, scalability and durability) can be easily improved by incorporating design patterns as aspects, without polluting the applications. There are also other case studies available in AO literature [11], [26] to reinforce the fact that non-functional features as aspects can be readily woven into software systems. Weaving concurrency patterns as aspects provide several advantages. However, the approach may expose limitations in certain scenarios. In this section, the advantages and limitations of the proposed approach are described based on its evaluation with that of conventional (OO) approach.

6.1. Comparison Evaluation of Conventional and Proposed Solution

From the case study, there are two main situations where the proposed solution (combination of aspects and design patterns) has been applied.

1. Refactoring an existing middleware RMI layer.
2. Enhancing new non-functional features such as persistence, encryption and concurrency features (locking a critical section, delayed retries, and thread pools).

From the implementations, demonstrated in section 5, it is clear that the two main situations implemented using the proposed solution has improved maintainability, serviceability, accountability, security, scalability and durability of the UIM. If the non-functional features have been implemented using the conventional OO approach, the primary system

code would have become polluted (scattered/intertwined) with the code pertaining to non-functional features. The isolated behavior of aspects prevents the primary system code from being polluted.

From the case study, following advantages and limitations are observed.

6.1.1. *Advantages*

The proposed approach (implementation using both patterns and aspects) reveals the following advantages for the UIM case study that could be very well observed in similar systems:

6.1.1.1. *Enhances Maintainability*

In the proposed approach, the original code need not (or minimally) be modified with the addition of non-functional features. In the present research, AO refactoring of RMI middleware implementation using the collaborator mediator design pattern has been examined. In the conventional approach, the middleware implementation layer affects almost all the non-trivial classes in the UIM system. For example, consider RMI version of Server class implemented in conventional approach (listed in Appendix- B.1). In that, almost all methods *throws RemoteException* (a tight coupling with the RMI middleware). In the conventional approach, the repeated *throws RemoteException*, is not further refactorable as they could not be extracted into one shared section. But with AO refactoring, all the *throws RemoteException* code is extracted into a single section in a separate aspect

(listed in Table 5.1). In addition, the other RMI code is also extracted into the aspect. With this aspect in place, the primary functional code is much cleaner and decoupled when compared to the conventional approach. Even if the middleware is changed to a different one, modifications are required only to the aspect and the primary functional code is oblivious about the new changes. Hence, the **maintainability**⁹ of the system is improved.

6.1.1.2. Improves Serviceability

The UIM case study recommends persisting client profile to improve serviceability. If the persistence mechanism has been implemented using the conventional approach, the code pertaining to persistence could be written in a separate method and could be called by all other methods that needs to write, soon after change(s) are made to client profile(s). The result of the conventional approach is that the call to the persistence method would have been spread across multiple methods in one or more classes. Whereas, the Decorator Wrapper aspect (illustrated in Figure 5.2 and Figure 5.3) implemented using the proposed solution improves system **serviceability**¹⁰ in an efficient and modularized manner preventing the primary functional code getting scattered/intertwined.

⁹ The idea behind the collaborator mediator aspect could be extended for other systems to improve maintainability. For example, in a distributed system, a different middleware implementation could benefit from the aspect by customizing according to system requirements.

¹⁰ The idea behind the decorator wrapper aspect could be extended for other systems to improve serviceability. For example, in a new system, when storing information into files are required, the system could benefit from the aspect by customizing according to its requirements.

6.1.1.3. *Enhances Accountability*

Based on the serviceability feature, the case study insists to lock the code pertaining to serviceability, as concurrent users trying to update their profile do not overwrite other's data. If locking the identified critical section has been implemented using the conventional approach, code for locking (upon entry of a critical section) and code for unlocking (upon exit of a critical section) would have been repeated for each critical section within the primary functional modules. Whereas the Code Locking aspect (Figure 5.4 and Figure 5.5) implemented using the proposed solution considerably reduces the amount of code and prevents the locking code from spreading/intertwining across primary functional modules. Even if the locking application programmable interface (API) changes to a different one, changes are required only to the aspects. Therefore, it is clear that the proposed solution improves system **accountability**¹¹ in an isolated manner without polluting the primary functional code.

6.1.1.4. *Increases Security*

The UIM case study recommends encrypting `userId` and password to secure the system. If encryption has been implemented using the conventional approach, *encrypt* method of the

¹¹ The idea behind the code locking aspect could be extended for other systems to improve accountability. For example, in a new system, when mutual exclusion of an object is required, the system could benefit from the aspect by customizing according to its requirements.

PBEencrypter class (listed in Figure 5.7) would have been called by each method that gets the input, `userId` and `password` data from the user. As UIM collects the `userId` & `password` in many places including registration, login, unregistration and adding a new friend (requires friend's `userId`), the *encrypt* method needs to be called individually on the `userId` and `password` data to encrypt them. In addition, the returned encrypted string from the *encrypt* method needs to be stored in additional variables and these variables have to be passed all along. Care should be taken that the encrypted value is passed over the unencrypted ones. Therefore, in the conventional approach, significant amount of encryption code is scattered across many modules. Whereas the State aspect (Figure 5.6) implemented using the proposed solution considerably reduces the amount of code and prevents the encryption code from spreading/intertwining across primary functional modules. Therefore, it is clear that the proposed solution improves system **security**¹² in a modular way without polluting the primary functional code.

6.1.1.5. Promotes System Scalability

To increase the performance of the UIM system, the case study recommends performing low priority tasks using separate threads asynchronously. However, if the Thread Pool pattern has been implemented using the conventional approach, the primary system would become complex with the scattered/intertwined implementation of asynchronous

¹² The idea behind the state aspect could be extended for other systems to improve security. For example, in a new system, encryption of any confidential data like messages could benefit from the aspect by customizing according to system requirements.

updateStatus, which is a low priority task. On the other hand, the proposed solution isolates the scattered/intertwined code into a Thread Pool aspect (listed in Figure 5.13). The same principle implemented in the previously described aspects such as code locking and persistence applies here. Thus, adopting the proposed approach reduces system complexities and increases system **scalability**¹³ as new threads created update the current status of user(s) and thus the system scales to handle more user requests.

6.1.1.6. Increases Durability

In typical client-server architecture, the client and server communicate across a distributed network. The UIM case study recommends retrying communications automatically, when the communication fails. However, if the Delayed Retry pattern has been implemented using the OOP approach, the primary system would become complex with the scattered/intertwined implementation for each remote method call. On the other hand, the proposed solution isolates the scattered/intertwined code into a delayed retry aspect (listed in Figure 5.15 and Figure 5.16). The same principle implemented in the previously described aspects such as code locking and persistence applies here. Thus, adopting the proposed solution increases system **durability**¹⁴.

¹³ The idea behind the thread pool aspect could be extended for other systems to improve scalability. For example, in a concurrent system, when asynchronous communication is preferred over synchronous, the system could benefit from the aspect by customizing according to its requirements.

¹⁴ The idea behind the delayed retry aspect could be extended for other systems to improve durability. For example, in a distributed system, when automated delayed retries are required, the system could benefit from the aspect by customizing it.

6.1.1.7. Increases Code Reusability

Because aspects implementing non-functional features are held isolated, they could be easily reused (or plugged in without requiring much or any modifications) within the same or different applications. The degree of reusability of aspects is high compared to that of OOP modules, where concerns (non-functional features) can be intertwined with primary functionalities. For example, the abstract Code Locking aspect could be plugged into another system without making any changes. All that is needed is a concrete Code Locking aspect to define the exact pointcuts for the new system. In the same way, abstract aspects like logging aspect (listed in Figure 2.3), could be plugged into another system without making any changes (not even the pointcuts need to be changed). Therefore, degree of **reusability** of aspects is higher compared to classes in conventional approach.

6.1.1.8. Increases Code Readability & Understandability

Because in the proposed approach, aspects and primary functional code are held isolated, the **readability** and therefore **understandability** of the entire system is improved that could be demonstrated by all the aspects illustrated in this present research.

To summarize, aspect-oriented programming in conjunction with design patterns and refactoring techniques reduces complexity of the system and promotes code reusability, readability, and understandability.

6.1.2. *Limitations*

The following limitations experienced in UIM could be very well observed in similar systems.

In the case study, the join point of Code Locking Aspect needs to capture only a few lines of system code responsible for synchronizing the critical section. However, because the join points could not capture just that small portion of the code, the entire content holder (method/constructor) had to be captured. In such situations that require lower granularity, applying aspect advice to the entire method might be over-killing and may even result in additional overhead. In the case study, because the source code of UIM system was available the critical section was isolated in its own method, negating the disadvantage. However, for legacy systems the source code may not be available. Even if the source code is available, making the changes may not be that simple.

The Figure 4.4, is modified (with the addition of image icons for each feature) from the original versions of UIM to increase its usability. On the other hand using aspects for introducing icons in the UIM GUI buttons would be too cumbersome, as pointcuts have to capture each button creation jointpoint along with its name and parameters. There needs to be one pointcut for each button and an advice to add this new image icon. A sys-

tem with numerous components (buttons) will require many icons and it is not feasible to implement this feature using aspects as the OO way of implementing would be simpler.

Even though the proposed approach, significantly improves readability when compared to the conventional approaches, in situations when there are many pointcuts addressing a single issue could degrade the aspect's readability.

7. Conclusions and Future Work

7.1. Conclusions

In the present research, aspects are used to implement reusable libraries for design patterns. Libraries of aspects are then integrated into a large-scale real world application, a universal instant messenger. These libraries could be easily reused in a different system requiring subtle or no modifications.

If concurrency features were implemented using conventional methodology (OOP), the primary system code would have to be polluted making it more complex and non-reusable. Moreover, the concurrency features would be intertwined and scattered along with the core functional code. If primary system happens to be a legacy system with no source code available then it is not even feasible to make any enhancements to it using the OOP methodology. All these limitations can be solved by adopting the suggested approach (implementation using both patterns and aspects).

The adopted approach also offers several advantages for software development and maintenance processes including improvements in maintainability, reusability, code readability and code understandability.

The present research also advocates that the initial versions of the new software systems be built focusing exclusively on core functional requirements and all the essential non-functional concerns be added using AOP and design patterns in the subsequent versions.

7.2. Future Work

AO approach provides opportunities to enhance several *ilities* of legacy systems that are otherwise not conceivable. In addition, the AOP approach permits to develop initial versions of new software system focusing only on core functional requirement. The case study reinforces that AOP is best suited for adding non-functional features to a software system. Analyzing advantages and limitations weaving caching and pooling related design patterns as aspects could be considered for future work.

Additional case studies can be made to analyze the advantages and limitations of adopting AOP in different software development processes. For example adopting AOP for testing processes would be useful to analyze.

References

Notes: This list follows the formatting guidelines of Association for Computing Machinery (ACM) research papers.

- [1] AspectJ: <http://www.eclipse.org/aspectj>.
- [2] CORBA: <http://www.corba.org>.
- [3] Aspect-Oriented Design Pattern Implementations: <http://www.cs.ubc.ca/~jan/AODPs/>
- [4] Remote Method Invocation: <http://java.sun.com/products/jdk/rmi/>
- [5] Remote Method Invocation Requirements:
<http://java.sun.com/docs/books/tutorial/rmi/implementing.html>
- [6] Refactoring : <http://www.refactoring.com/>
- [7] « illities » : <http://www.searchword.org/il/ilities.html>
- [8] Universal Instant Messenger: <http://sourceforge.net/>
- [9] Aspect-Oriented refactoring series:
<http://www.theserverside.com/articles/article.tss?l=AspectOrientedRefactoringPart1>
- [10] A.K. Mok and S.A. Ward. Distributed Broadcast Channel Access. *Computer Networks*, Vol. 3, pages 327-335, March 1985.
- [11] A. Rashid and R. Chitchyan. Persistence as an aspect. In *Proc. Of the 2nd International conference on Aspect-oriented software development*, pages 120-129, Boston, USA, 2003.

- [12] C. Zhang, D. Gao and H. Jacobsen. Towards just-in-time middleware architectures. In *Proc. Of the 4th International conference on Aspect-oriented software development*, pages 63-74, Chicago, USA, 2005.
- [13] D. C. Schmidt, M. Stal, H. Rohnert and F. Buschmann. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. Wiley Publishing Co., 2000.
- [14] D. Lea. *Concurrent programming in Java: Design Principles and Patterns*. Addison-Wesley, 2000.
- [15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [16] F. Cheng. Policies As Design And Implementation Artifacts For Non Functional Requirements. Master of Engineering Thesis, Carleton University, Ottawa, Canada, 2002.
- [17] F. Naso and A. Fernandez. Rendezvous. In *Proc. Of the 11th Pattern Languages of Programs Conference*, Chicago, USA, 2004.
- [18] I. Jacobsen, P. Wei. Aspect-Oriented Software Development. With Use Cases, 2005
- [19] I. Pyarali, T. H. Harrison, and D. C. Schmidt. *Asynchronous Completion Token: an Object Behavioral Pattern for Efficient Asynchronous Event Handling in Pattern Languages of Program Design*. Addison-Wesley, 1997.
- [20] J. Hannemann and G. Kiczales. Design Pattern Implementation in Java and AspectJ. *Proceedings of the 17th Annual ACM conference on Object-Oriented Pro-*

- gramming, Systems, Languages, and Applications (OOPSLA)*, pages 161-173, November 2002.
- [21] J. Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2004.
 - [22] J. L. Ortega-Arjona. The Shared Resource Pattern. In *Proc. Of the 10th Pattern Languages of Programs Conference*, Chicago, USA, 2003.
 - [23] J. Thomas, M. Young, K. Brown and A. Glover. *Java Testing Patterns*. Wiley, 2004.
 - [24] L. Kleinrock and F. Tobagi. Random Access Techniques for Data Transmission over Packet-Switched Radio Channels. In *Proc. National Computer Conference*, pages 187-201, Anaheim, USA, 1975.
 - [25] L. Roberts. Extensions of Packet Communication Technology to a Hand Held Personal Terminal. In *Proc. Of the Spring Joint Computer Conference*, pages 295-298, Atlantic City, USA, 1972.
 - [26] L.T. Cheng, J. Patterson, S. L. Rohall, S. Hupfer, S. Ross. Weaving a social fabric into existing software. In *Proc. Of the 4th International conference on Aspect-oriented software development*, pages 147-158, Chicago, USA, 2005.
 - [27] M. Grand. *Java Enterprise Design Patterns: Patterns in Java Volume 3*. Wiley, 2001.
 - [28] N. Abramson. Development of the ALOHANET. In *IEEE Tran. On Information Theory*, Vol. IT-31, pages 119-123, March 1985.

- [29] P. E. McKenney. Selecting locking primitives for parallel programming. In *Proc. Of Communications of the ACM*, Vol. 39 (Issue 10), pages 75-82, New York, USA, 1996.
- [30] P. Karn. MACA—A New Channel Access Protocol for Packet Radio. In *Proc. Of the 9th Computer Networking Conference*, pages 134-140, London, Canada, 1990.
- [31] R. Laddad. AspectJ In Action. *Manning Publications*, 2003.
- [32] R. Silaghi, A. Strohmeier. Parallax – An Aspect-Enabled Framework for Plugin-Based Model Driven Architecture Refinements Towards Middleware. *Book Chapter in “Model-Driven Software Development”, Volume II of “Research and Practice in Software Engineering”*, Switzerland, May 2005.

Appendix A AspectJ V1.1 Keywords (Quick Reference)

AspectJ 1.1 Quick Reference

Aspects *at top-level or static in types*

aspect *A* { ... }
defines the aspect *A*
privileged aspect *A* { ... }
A can access private fields
aspect *A* **extends** *B* **implements** *I*, *J* { ... }
B is a class or abstract aspect, *I* and *J* are interfaces
aspect *A* **perflow**(*call(void Foo.m())*) { ... }
an instance of *A* is instantiated for every control flow through calls to *m()*

general form:
[**privileged**] [*Modifiers*] **aspect** *Id*
[**extends** *Type*] [**implements** *TypeList*] [*PerClause*]
{ *Body* }
where *PerClause* is one of
pertarget (*Pointcut*)
perthis (*Pointcut*)
perflow (*Pointcut*)
perflowbelow (*Pointcut*)
issingleton

Pointcut definitions *in types*

private pointcut *pc()* : *call(void Foo.m())* ;
a pointcut visible only from the defining type
pointcut *pc(int i)* : *set(int Foo.x) && args(i)* ;
a package-visible pointcut that exposes an *int*.
public abstract pointcut *pc()* ;
an abstract pointcut that can be referred to from anywhere.
abstract pointcut *pc(Object o)* ;
an abstract pointcut visible from the defining package. Any pointcut that implements this must expose an *Object*.

general form:
abstract [*Modifiers*] **pointcut** *Id* (*Formals*) ;
[*Modifiers*] **pointcut** *Id* (*Formals*) : *Pointcut* ;

Advice declarations *in aspects*

before () : *get(int Foo.y)* { ... }
runs before reading the field *int Foo.y*
after () **returning** : *call(int Foo.m(int))* { ... }
runs after calls to *int Foo.m(int)* that return normally
after () **returning** (*int x*) : *call(int Foo.m(int))* { ... }
same, but the return value is named *x* in the body
after () **throwing** : *call(int Foo.m(int))* { ... }
runs after calls to *m* that exit abruptly by throwing an exception
after () **throwing** (*NotFoundException e*) : *call(int Foo.m(int))* { ... }
runs after calls to *m* that exit abruptly by throwing a *NotFoundException*. The exception is named *e* in the body
after () : *call(int Foo.m(int))* { ... }
runs after calls to *m* regardless of how they exit
before(*int i*) : *set(int Foo.x) && args(i)* { ... }
runs before field assignment to *int Foo.x*. The value to be assigned is named *i* in the body
before(*Object o*) : *set(* Foo.*) && args(o)* { ... }
runs before field assignment to any field of *Foo*. The value to be assigned is converted to an object type (*int* to *Integer*, for example) and named *o* in the body

int **around** () : *call(int Foo.m(int))* { ... }
runs instead of calls to *int Foo.m(int)*, and returns an *int*. In the body, continue the call by using **proceed**(), which has the same signature as the around advice.
int **around** () **throws** *IOException* : *call(int Foo.m(int))* { ... }
same, but the body is allowed to throw *IOException*
Object **around** () : *call(int Foo.m(int))* { ... }
same, but the value of **proceed**() is converted to an *Integer*, and the body should also return an *Integer* which will be converted into an *int*

general form:
[**strictfp**] *AdviceSpec* [**throws** *TypeList*] : *Pointcut* { *Body* }
where *AdviceSpec* is one of
before (*Formals*)
after (*Formals*)
after (*Formals*) **returning** { (*Formal*) }
after (*Formals*) **throwing** { (*Formal*) }
Type **around** (*Formals*)

Special forms *in advice*

thisJoinPoint
reflective information about the join point
thisJoinPointStaticPart
the equivalent of **thisJoinPoint.getStaticPart()**, but may use fewer resources.
thisEnclosingJoinPointStaticPart
the static part of the join point enclosing this one.

proceed (*Arguments*)
only available in **around** advice. The *Arguments* must be the same number and type as the parameters of the advice.

Inter-type Member Declarations *in aspects*

int *Foo* . *m* (*int i*) { ... }
a method *int m(int)* owned by *Foo*, visible anywhere in the defining package. In the body, **this** refers to the instance of *Foo*, not the aspect.
private *int* *Foo* . *m* (*int i*) **throws** *IOException* { ... }
a method *int m(int)* that is declared to throw *IOException*, only visible in the defining aspect. In the body, **this** refers to the instance of *Foo*, not the aspect.
abstract *int* *Foo* . *m* (*int i*) ;
an abstract method *int m(int)* owned by *Foo*
Point . **new** (*int x, int y*) { ... }
a constructor owned by *Point*. In the body, **this** refers to the new *Point*, not the aspect.
private static *int* *Point* . *x* ;
a static *int* field named *x* owned by *Point* and visible only in the declaring aspect
private *int* *Point* . *x* = *foo()* ;
a non-static field initialized to the result of calling *foo()*. In the initializer, **this** refers to the instance of *Foo*, not the aspect.

general form:
[*Modifiers*] *Type* *Type* . *Id* (*Formals*)
[**throws** *TypeList*] { *Body* }
abstract [*Modifiers*] *Type* *Type* . *Id* (*Formals*)
[**throws** *TypeList*] ;
[*Modifiers*] *Type* . **new** (*Formals*)
[**throws** *TypeList*] { *Body* }
[*Modifiers*] *Type* *Type* . *Id* [= *Expression*] ;

Other Inter-type Declarations *in aspects*

declare parents : *C extends D*;
declares that the superclass of *C* is *D*. This is only legal if *D* is declared to extend the original superclass of *C*.

declare parents : *C implements I, J*;
C implements *I* and *J*

declare warning : *set(* Point.*) && !within(Point) : "bad set"*;
the compiler warns "bad set" if it finds a set to any field of *Point* outside of the code for *Point*

declare error : *call(Singleton.new(..)) : "bad construction"*;
the compiler signals an error "bad construction" if it finds a call to any constructor of *Singleton*

declare soft : *IOException : execution(Foo.new(..))*;
any *IOException* thrown from executions of the constructors of *Foo* are wrapped in *org.aspectj.SoftException*

declare precedence : *Security, Logging, **;
at each join point, advice from *Security* has precedence over advice from *Logging*, which has precedence over other advice.

general form

```
declare parents : TypePat extends Type ;
declare parents : TypePat implements TypeList ;
declare warning : Pointcut : String ;
declare error : Pointcut : String ;
declare soft : Type : Pointcut ;
declare precedence : TypePatList ;
```

Primitive Pointcuts

call (*void Foo.m(int)*)
a call to the method *void Foo.m(int)*

call (*Foo.new(..)*)
a call to any constructor of *Foo*

execution (** Foo.*(..) throws IOException*)
the execution of any method of *Foo* that is declared to throw *IOException*

execution (*!public Foo.new(..)*)
the execution of any non-public constructor of *Foo*

initialization (*Foo.new(int)*)
the initialization of any *Foo* object that is started with the constructor *Foo(int)*

preinitialization (*Foo.new(int)*)
the pre-initialization (before the *super* constructor is called) that is started with the constructor *Foo(int)*

staticinitialization (*Foo*)
when the type *Foo* is initialized, after loading

get (*int Point.x*)
when *int Point.x* is read

set (*!private * Point.**)
when any non-private field of *Point* is assigned

handler (*IOException+*)
when an *IOException* or its subtype is handled with a catch block

adviceexecution()
the execution of all advice bodies

within (*com.bigboxco.**)
any join point where the associated code is defined in the package *com.bigboxco*

withincode (*void Figure.move()*)
any join point where the associated code is defined in the method *void Figure.move()*

withincode (*com.bigboxco.*.new(..)*)
any join point where the associated code is defined in any constructor in the package *com.bigboxco*.

cflow (*call(void Figure.move())*)
any join point in the control flow of each call to *void Figure.move()*. This includes the call itself.

cflowbelow (*call(void Figure.move())*)
any join point below the control flow of each call to *void Figure.move()*. This does not include the call

if (*Tracing.isEnabled()*)
any join point where *Tracing.isEnabled()* is *true*. The boolean expression used can only access static members, variables bound in the same pointcut, and *thisJoinPoint* forms.

this (*Point*)
any join point where the currently executing object is an instance of *Point*

target (*java.io.InputPort*)
any join point where the target object is an instance of *java.io.InputPort*

args (*java.io.InputPort, int*)
any join point where there are two arguments, the first an instance of *java.io.InputPort*, and the second an *int*

args (***, *int*)
any join point where there are two arguments, the second of which is an *int*.

args (*short, ..., short*)
any join point with at least two arguments, the first and last of which are *shorts*

Note: any position in **this**, **target**, and **args** can be replaced with a variable bound in the advice or pointcut.

general form:

```
call(MethodPat)
call(ConstructorPat)
execution(MethodPat)
execution(ConstructorPat)
initialization(ConstructorPat)
preinitialization(ConstructorPat)
staticinitialization(TypePat)
get(FieldPat)
set(FieldPat)
handler(TypePat)
adviceexecution()
within(TypePat)
withincode(MethodPat)
withincode(ConstructorPat)
cflow(Pointcut)
cflowbelow(Pointcut)
if(Expression)
this(Type | Var)
target(Type | Var)
args(Type | Var, ...)
```

where *MethodPat* is:
[*ModifiersPat*] *TypePat* [*TypePat* .] *IdPat* (*TypePat* | ...)
[*throws ThrowsPat*]

ConstructorPat is:
[*ModifiersPat*] [*TypePat* .] *new* (*TypePat* | ...)
[*throws ThrowsPat*]

FieldPat is:
[*ModifiersPat*] *TypePat* [*TypePat* .] *IdPat*

TypePat is one of:
IdPat [+] { [...]
! *TypePat*
TypePat && *TypePat*
TypePat || *TypePat*
(*TypePat*)

Acknowledgement: Sincere thanks for the quick reference [31].

Appendix B Additional Classes and aspects for Collaborator Mediator Pattern

B.1 RMI implementation of Server in OOP

The RMI implementation of Server class is presented below:

```
public class Server extends UnicastRemoteObject implements RMIServerInterface {

    // ref to UimRegistry
    private RemoteUimRegistryInterface RemoteRegistryInterface;

    // key: agentInfo(Copy of Agent), value: friends of agentInfos
    private Hashtable buddyTable;

    // by storing the password outside the AgentInfo object,
    // we don't run the risk of disclosing passwords to other buddies
    // when sending them AgentInfo objects
    // however this makes the code more complex...
    private Hashtable passwordTable;

    public Server() throws Exception
    {
        UimRegistry r = new UimRegistry();
        buddyTable = new Hashtable();
        passwordTable = new Hashtable();
        Naming.bind("uimServer", this);
        RemoteRegistryInterface = (RemoteUimRegistryInterface) Naming.lookup("rmi://localhost/registry");
    }

    /** check whether agentid already exists in buddyTable, if so throw an exception
     * otherwise create new entry in buddyTable */
    public void subscribe (String agentId, int password) throws RemoteException, AgentIdAlreadyInUseException
    {
        //Check for null
        if (agentId == null) throw new NullPointerException("Expecting Agent id");
        // I am only happy if I can *not* find the agentId in the buddyTable!
        try {
            AgentInfo ai = getRealKey(agentId);
            throw new AgentIdAlreadyInUseException();
        }
        catch (NoSuchAgentException e) {
            AgentInfo ai2 = new AgentInfo (agentId);
            buddyTable.put(ai2, new HashSet()); //hashset is the simplest implementation of the set interface
            passwordTable.put(agentId, new Integer(password));
        }
        System.out.println("Subscription successful");
    }

    /** remove entry in buddyTable */
    public void unsubscribe (String agentId, int password) throws RemoteException, InvalidPasswordException,
```

```

NoSuchAgentException
{
    AgentInfo ai = getRealKey(agentId);
    if (!passwordValid(agentId, password)) throw new InvalidPasswordException();
    //should I make sure I'm logged out before I remove myself?
    // the assumption is that I can only unsubscribe when I'm logged out but still...
    // I wouldn't want people to try to IM me if I don't exist!
    // worth doing it since this is not an operation that is frequently used anyway...
    logout(agentId, password); //ideally there should be a logout(agentInfo) method...
    buddyTable.remove(ai);
    passwordTable.remove(ai);
}

public AgentInfo addFriend(String agentId, int password, String otherAgentId) throws RemoteException,
    InvalidPasswordException, NoSuchAgentException
{
    AgentInfo ai = getRealKey(agentId);
    AgentInfo ai2 = getRealKey(otherAgentId);

    // check password
    if (!passwordValid(agentId, password)) throw new InvalidPasswordException();
    //add the *ref* to complete agentInfo to the buddy list and return it if success.
    ((Set) buddyTable.get(ai)).add(ai2);
    return ai2;
}

public AgentInfo removeFriend(String agentId, int password, String otherAgentId) throws RemoteException,
    InvalidPasswordException, NoSuchAgentException
{
    //check both agent ids
    AgentInfo ai = getRealKey(agentId);
    AgentInfo ai2 = getRealKey(otherAgentId);

    // check password
    if (!passwordValid(agentId, password)) throw new InvalidPasswordException();
    if ( ((Set) buddyTable.get(ai)).remove(ai2) ) return ai2; else return null;
}

public Set login(String agentId, int password, String agentAddress) throws RemoteException,
    InvalidPasswordException, NoSuchAgentException
{
    // check agent id.
    AgentInfo ai = getRealKey(agentId);
    // check password
    if (!passwordValid(agentId, password)) throw new InvalidPasswordException();
    // logon!
    changeStatus(agentId, password, Agent.AVAILABLE);
    // send back friend list with up to date status info.
    return (Set) buddyTable.get(ai);
}

public void logout(String agentId, int password) throws RemoteException,
    InvalidPasswordException, NoSuchAgentException
{
    // same as changing status to 0
    changeStatus(agentId, password, Agent.LOGGEDOFF);
}

public void changeStatus(String agentId, int password, int status) throws RemoteException,
    InvalidPasswordException, NoSuchAgentException
{
    // check agent id
    AgentInfo ai = getRealKey(agentId);

    // check password

```

```

        if (!passwordValid(agentId, password)) throw new InvalidPasswordException();
        // send update to all the agent's buddies.
        // tell logged on guys who have him in friend list that he's logged out.
        Iterator it = buddyTable.keySet().iterator();
        while (it.hasNext()) {
            AgentInfo key = (AgentInfo) it.next();
            if ( ((Set) buddyTable.get(key)).contains(ai) && !(key.getStatus()==Agent.LOGGEDOFF) ) {
                try {
                    CallbackClientInterface a = (CallbackClientInterface)
RemoteRegistryInterface.lookup(key.getAgentId());
                    if (a != null) { // == null can happen if I'm my own friend
                        a.handleStatusChange(new StatusChangeEvent(agentId, status) );
                    }
                }
                catch (RemoteException e) {System.out.println("remote exception on server trying handle status change");
                }
            }
        }
        // update status only now
        // (to avoid sending updates to myself (if I'm my own friend) before I'm ready)
        ai.setStatus(status);
        System.out.println("New status for " + ai);
        System.out.println("Buddy Table after status change: " + buddyTable);
    }

    /**
     * cleanup
     */
    public void shutDown() {
        try {
            Naming.unbind("server");
        }
        catch (RemoteException e) {}
    }

    /** find the key that corresponds to the agent id. */
    private AgentInfo getRealKey(String agentId) throws NoSuchAgentException {
        AgentInfo key = null;
        Iterator it = buddyTable.keySet().iterator();
        while (it.hasNext()) {
            key = (AgentInfo) it.next();
            if (key.getAgentId().equals(agentId)) return key;
        }
        throw new NoSuchAgentException();
    }

    private boolean passwordValid (String agentId, int password) {
        if ( ((Integer) (passwordTable.get(agentId))).intValue() == password)
            return true;
        return false;
    }

    public static void main (String[] args) throws RemoteException {
        Server s = new Server();
    }
}

```

B.2 Collaborator mediator aspect for Agent

The collaborator mediator aspect for the Agent collaborator on the client side is presented below:

```
import java.net.MalformedURLException;
import java.rmi.*; import java.rmi.server.UnicastRemoteObject;
import java.util.*;
public aspect RmiAgentAspect {

    declare parents: Agent extends UnicastRemoteObject;
    declare parents: Agent implements CallbackClientInterface;

    public static RMIServerInterface Agent.remoteServerInterface;
    public static RemoteUimRegistryInterface Agent.RemoteRegistryInterface;
    public static String Agent.agentName4;

    public static void Agent.contactServer() throws MalformedURLException, ConnectException,
        NotBoundException, RemoteException {
        System.out.println("I am connecting to server.....");
        if (Agent.remoteServerInterface == null) {
            Agent.remoteServerInterface = (RMIServerInterface) Naming.lookup("rmi://" + Agent.serverAddress +
                "/uimServer");
        }
        Agent.RemoteRegistryInterface = (RemoteUimRegistryInterface) Naming.lookup("rmi://" +
            Agent.serverAddress + "/uim21.registry");
    }

    public pointcut subscribeToServer(String agentName, String agentId, String password)
        : execution(public boolean *.register(String, String, String)) && args(agentName, agentId, password);

    boolean around(String agentName, String agentId, String password)
        throws AgentIdAlreadyInUseException, NoSuchAgentException :
        subscribeToServer(agentName, agentId, password)
    {
        boolean serverAvailabe = false;
        try {
            Agent.contactServer();
            Agent.remoteServerInterface.subscribe(agentName, agentId, password);
        }
        catch (Exception ex){ex.printStackTrace();}
        serverAvailabe = true;
        return serverAvailabe;
    }

    public pointcut unsubscribeFromServer(String agentId, String password)
        : execution(public void *.unregister(String, String)) && args(agentId, password);

    void around(String agentId, String password)
        throws InvalidPasswordException, NoSuchAgentException :
        unsubscribeFromServer(agentId, password)
    {
        try {
            Agent.contactServer();
            Agent.remoteServerInterface.unsubscribe(agentId, password);
        }
        catch (Exception ex){ex.printStackTrace();}
    }

    public pointcut loginAgentToServer(String agentId, String password)
```

```

: execution(public Agent Agent.login(String, String)) && args(agentId, password);

Agent around(String agentId, String password)
    throws InvalidPasswordException, NoSuchAgentException :
    loginAgentToServer(agentId, password)
{
    Agent me = null;
    HashSet bl = null;

    try {
        Agent.contactServer();
        bl = (HashSet) Agent.remoteServerInterface.login(agentId, password, "local");
        Agent.agentName4 = Agent.remoteServerInterface.getAgentName(agentId);
        me = new Agent(Agent.agentName4, agentId, password, bl);
        Agent.RemoteRegistryInterface.register(agentId, me);
        me.setStatus(Agent.AVAILABLE);
    }
    catch (Exception ex){ex.printStackTrace();}
    return me;
}

public pointcut loginToServer(String agentId, String password)
: execution(public Agent HumanAgent.login(String, String)) && args(agentId, password);

Agent around(String agentId, String password)
    throws InvalidPasswordException, NoSuchAgentException :
    loginToServer(agentId, password)
{
    HumanAgent me = null;
    HashSet bl = null;

    try {
        Agent.contactServer();
        bl = (HashSet) Agent.remoteServerInterface.login(agentId, password, "local");
        Agent.agentName4 = Agent.remoteServerInterface.getAgentName(agentId);
        me = new HumanAgent(Agent.agentName4, agentId, password, bl);
        Agent.RemoteRegistryInterface.register(agentId, me);
        me.setStatus(Agent.AVAILABLE);
    }
    catch (Exception ex){ex.printStackTrace();}
    return me;
}

public pointcut logoutFromServer(String agentId)
: execution(public void Agent.logout(String)) && args(agentId);

void around(String agentId)
    throws InvalidPasswordException, NoSuchAgentException :
    logoutFromServer(agentId)
{
    proceed(agentId);

    try {
        Agent.remoteServerInterface.logout(agentId);
        Agent.RemoteRegistryInterface.unregister(agentId);
    }
    catch (Exception ex){ex.printStackTrace();}
}

public pointcut changeMyStatus(Agent agent, int status)
: execution(public void *.changeStatus(Agent, int)) && args(agent, status);

void around(Agent agent, int status)
    throws InvalidPasswordException, NoSuchAgentException :

```



```

    changeMyStatus(agent, status)
    {
        proceed(agent, status);

        try {
            Agent.remoteServerInterface.changeStatus(agent.getAgentId(), agent.getPassword(), status);
        }
        catch (Exception ex){ex.printStackTrace();}
    }

    public pointcut addBuddy(Agent agent, String buddyId)
    : execution(public void *.addFriend(Agent, String)) && args(agent, buddyId);

    void around(Agent agent, String buddyId)
    throws InvalidPasswordException, NoSuchAgentException :
        addBuddy(agent, buddyId)
    {
        try {
            AgentInfo buddyInfo = Agent.remoteServerInterface.addFriend(agent.getAgentId(),
agent.getPassword(),
                                                                    buddyId);
            agent.getBuddyList().add(buddyInfo);
        }
        catch (Exception ex){ex.printStackTrace();}
    }

    public pointcut deleteBuddy(Agent agent, String buddyId)
    : execution(public void *.removeFriend(Agent, String)) && args(agent, buddyId);

    void around(Agent agent, String buddyId)
    throws InvalidPasswordException, NoSuchAgentException :
        deleteBuddy(agent, buddyId)
    {
        try {
            AgentInfo buddyInfo = Agent.remoteServerInterface.removeFriend(agent.getAgentId(),
agent.getPassword(), buddyId);
            if (buddyInfo != null) agent.getBuddyList().remove(buddyInfo);
        }
        catch (Exception ex){ex.printStackTrace();}
    }

    public pointcut sendMessageToBuddy(String buddyId, String message)
    : execution(public void *.sendMessage(String, String)) && args(buddyId, message);

    void around(String buddyId, String message) :
        sendMessageToBuddy(buddyId, message)
    {
        InstantMessage im = new InstantMessage(Agent.agentName4, buddyId, message);
        try {

            ((CallbackClientInterface)
Agent.RemoteRegistryInterface.lookup(buddyId)).handleInstantMessage(im);
        }
        catch (Exception ex){ex.printStackTrace();}
        System.out.println("message to: " + im.getReceiver() + ": " + im.getMessage());
    }
}

private static aspect SofteningRemoteException{
    declare soft: java.rmi.RemoteException
    : (call (* Agent.*(..))
}

```

Appendix C Classes referred in Thread Pool Aspect

C.1 Class - AsyncClientCallBack

```
public class AsyncClientCallBack implements Serializable {
    private int fNumNotifiers;
    private int fPriority;
    private Vector fCallbacks = new Vector();
    private Vector fNotifiers = new Vector();
    private long fMaxCalltime = TEN_SECONDS;
    private Thread fSweeper = new SweeperThread();

    public static final long TEN_SECONDS = 10000;
    public static final long THIRTY_SECONDS = 30000;
    public static final long SIXTY_SECONDS = 60000;
    public static int SWEEP_RATE = 4000;

    public AsyncClientCallBack(int priority, int num) {
        if (priority < Thread.MIN_PRIORITY || priority > Thread.MAX_PRIORITY) {
            throw new IllegalArgumentException("Invalid thread priority: " +
                priority);
        }
        fNumNotifiers = num;
        fPriority = priority;
        fSweeper.start();
    }

    public synchronized void start() {
        NotifyThread n;
        if (fNumNotifiers == 0) return;
        for (int i = fNumNotifiers; i > 0; i--) {
            n = new NotifyThread(i, fPriority, this);
            fNotifiers.addElement(n);
        }
    }

    public synchronized void stop() {
        Enumeration e = fNotifiers.elements();
        NotifyThread n;
        while (e.hasMoreElements()) {
            n = (NotifyThread)e.nextElement();
            n.timeToStop();
        }
        fNotifiers.removeAllElements();
    }

    public void doClientUpdate(ClientUpdate c) {
        doClientUpdate (c, null, null, -1);
    }

    public void doClientUpdate () {
        doClientUpdate (null, null, -1);
    }

    public void doClientUpdate (String arg, String arg2, int arg3) {
        RequestClientAsyncCallBack cbr = new RequestClientAsyncCallBack(arg, arg2, arg3);
        doClientUpdate (cbr);
    }

    public void doClientUpdate ClientUpdate c, String arg, String arg2, int arg3) {
        RequestClientAsyncCallBack cbr = new RequestClientAsyncCallBack(c, arg, arg2, arg3);
    }
}
```

```

        doClientUpdate (cbr);
    }

    public synchronized void doClientUpdate (RequestClientAsyncCallBack c) {
        fCallbacks.addElement(c);
        notify();
    }

    public int getNumThreads() {
        return fNumNotifiers;
    }

    public void setMaxCalltime(long t) {
        fMaxCalltime = t;
    }

    public long getMaxCalltime() {
        return fMaxCalltime;
    }

    void checkForTimeout(long curr, long max) {
        Enumeration e = fNotifiers.elements();
        while (e.hasMoreElements()) {
            NotifyThread n = (NotifyThread)e.nextElement();
            n.checkForTimeout(curr, max);
        }
    }

    public void waitForCallback(NotifyThread n) throws InterruptedException {
        RequestClientAsyncCallBack req = null;
        synchronized(this) {
            while (fCallbacks.size() == 0) {
                wait();
            }
            req = (RequestClientAsyncCallBack)fCallbacks.elementAt(0);
            fCallbacks.removeElementAt(0);
        }

        n.setRequest(req);
    }

    class SweeperThread extends Thread implements Serializable{
        SweeperThread() {
            setPriority(MIN_PRIORITY);
            setDaemon(true);
        }

        public void run() {
            while (true) {
                try {
                    sleep(SWEEP_RATE);
                } catch (InterruptedException ex) {}

                long max = fMaxCalltime;
                long curr = System.currentTimeMillis();

                checkForTimeout(curr, max);
            }
        }
    }
}

```

C.2 Interface – ClientUpdate

```
public interface ClientUpdate extends ja.rmi.Remote{
    public void CallBack(NotifyThread n, String arg1, String arg2, int arg3) throws
        RemoteException;
}
```

C.3 Class – RequestClientAsyncCallBack

```
public class RequestClientAsyncCallBack implements java.io.Serializable{
    private ClientUpdate fToCall;
    private String fArg;
    private int fArg3;

    public RequestClientAsyncCallBack(ClientUpdate c, String arg, String arg2, int arg3) {
        fToCall = c;
        fArg = arg;
        fArg2 = arg2;
        fArg3 = arg3;
    }

    public RequestClientAsyncCallBack(String arg, String arg2, int arg3) {
        fArg = arg;
        fArg2 = arg2;
        fArg3 = arg3;
    }

    /**
     * calls the ClientUpdate's callBack method
     */

    public void callback(NotifyThread n) throws RemoteException {
        fToCall.CallBack(n, fArg, fArg2, fArg3);
    }
}
```

C.4 Class – NotifyThread

```
public class NotifyThread extends Thread implements java.io.Serializable{
    private boolean fStopNow;
    private long fStartedCallback;
    private AsyncClientCallBack fList;
    private RequestClientAsyncCallBack fToCallback;
    private boolean fIsCallbackActive = false;

    NotifyThread(int id, int pri, AsyncClientCallBack list) {
        super("NotifyThread "+pri+"-"+id);
        setPriority(pri);
        fList = list;

        start();
    }

    public void run() {
        while (!fStopNow) {
```

```

        try {
            fList.waitForCallback(this);
            resetCallbackTime();
            flsCallbackActive = true;
            fToCallback.callback(this);
        } catch (InterruptedException ex) {
            if (fStopNow) return;
            System.out.println("ClientUpdate exceeded maximum time or time to stop");
        } catch (Exception ex) {
            System.err.println("Unexpected callback error: " + ex);
            ex.printStackTrace(System.err);
        } finally {
            flsCallbackActive = false;
        }
    }
}

/**
 * to set the next clientUpdate request
 * to execute.
 */
public void setRequest(RequestClientAsyncCallBack c) {
    fToCallback = c;
}

/**
 * Sets the stop flag which will cause the run() method to terminate after
 * completing current activity
 */
public void timeToStop() {
    fStopNow = true;
    interrupt();
}

/**
 * Gets the time since the last clientUpdate was started based on the current
 * system time.
 */
public long getElapsedCallbackTime() {
    return getElapsedCallbackTime(System.currentTimeMillis());
}

/**
 * Sets the time since the last clientUpdate was started based on the system time
 * passed into the method.
 */
public long getElapsedCallbackTime(long curr) {
    return curr - fStartedCallback;
}

/**
 * Gets whether this NotifyThread is currently executing a clientUpdate.
 */
public boolean isCallbackActive() {
    return flsCallbackActive;
}

/**
 * Resets the elapsed callback time to 0. This method should be called by
 * a ClientUpdate object before each method dispatch.
 */
public void resetCallbackTime() {
    fStartedCallback = System.currentTimeMillis();
}

```

```
final void checkForTimeout(long curr, long max) {  
    if (!isCallbackActive) {  
        return;  
    }  
  
    if (getElapsedCallbackTime(curr) > max) {  
        interrupt();  
    }  
}
```

Appendix D List Of Common Pointcuts

The following are some list of common pointcuts applied in the current research in various aspects.

Discrete Logging Aspect (Figure 2.3)

```
pointcut traceMethods() : (execution(* *.*(..) || execution(*.new(..)))  
    && !within(DiscreetLoggingAspect));
```

Collaborator Mediator Aspect (Table 5.1 and Appendix B.2)

```
public pointcut changeAgentStatusInTheirBuddyList  
    (String agentId, String password, int status)  
    : execution(public void Server.changeStatus(String, String, int)) && args(agentId,  
    password, status);  
  
public pointcut subscribeToServer(String agentName, String agentId, String password)  
    : execution(public boolean *.register(String, String, String)) && args(agentName,  
    agentId, password);  
  
public pointcut unSubscribeFromServer(String agentId, String password)  
    : execution(public void *.unregister(String, String)) && args(agentId, password);
```

```
public pointcut loginAgentToServer(String agentId, String password)
    : execution(public Agent Agent.login(String, String)) && args(agentId, password);
```

```
public pointcut logoutFromServer(String agentId)
    : execution(public void Agent.logout(String)) && args(agentId);
```

```
public pointcut changeMyStatus(Agent agent, int status)
    : execution(public void *.changeStatus(Agent, int)) && args(agent, status);
```

```
public pointcut addBuddy(Agent agent, String buddyId)
    : execution(public void *.addFriend(Agent, String)) && args(agent, buddyId);
```

```
public pointcut deleteBuddy(Agent agent, String buddyId)
    : execution(public void *.removeFriend(Agent, String)) && args(agent, buddyId);
```

```
public pointcut sendMessageToBuddy(String buddyId, String message)
    : execution(public void *.sendMessage(String, String)) && args(buddyId, message);
```

Decorator Wrapper Aspect (Figure 5.2 and Figure 5.3)

```
pointcut writeStorage() :
    execution(public void Server.subscribe(..))
    | | execution(public void Server.unsubscribe(..))
    | | execution(public AgentInfo Server.addFriend(..))
```



```
| | execution(public AgentInfo Server.removeFriend(..)) ;
```

```
pointcut readStorage() : call(Server.new());
```

Code Locking Aspect (Figure 5.5)

```
pointcut criticalSection () :
```

```
    execution(public void Server.subscribe(..))
```

```
    | | execution(public void Server.unsubscribe(..))
```

```
    | | execution(public AgentInfo Server.addFriend(..))
```

```
    | | execution(public AgentInfo Server.removeFriend(..)) ;
```

State Aspect (Figure 5.6)

```
pointcut encryptUserId(LifeCyclePanel lcg, String agentId):
```

```
    call(public void LifeCyclePanel.setAgentId(String))&& args(agentId)&&
```

```
    withincode(* LifeCyclePanel.*(..)) &&
```

```
    !withincode(* LifeCyclePanel.setAgentId(..))
```

```
    && target (lcg);
```

```
pointcut encryptPassword(LifeCyclePanel lcg, String password):
```

```
    call(public void LifeCyclePanel.setPassword(String))&& args(password)
```

```
    && withincode(* LifeCyclePanel.*(..)) &&
```

```
    !withincode(* LifeCyclePanel.setPassword(..))
```

```
    && target (lcg);
```

Thread Pool Aspect (Figure 5.13)

```
pointcut initThreadPool()
```

```
    : call(Server.new());
```

```
pointcut asyncChangeStatus(String agentId, String password, int status)
```

```
    : call(public void *.updateStatus(String, String, int)) &&
```

```
    args(agentId, password, status);
```

Delayed Retry Aspect (Figure 5.16)

```
public pointcut retryWhenNotAccessible
```

```
    (String agentName, String agentId, String password)
```

```
    : call(public boolean *.register(String, String, String)) &&
```

```
    args(agentName, agentId, password);
```