5-2014

Improving Reuse and Maintainability of Communication Software With Conversation-Aware Aspects

Ali Raza
Utah State University

IMPROVING REUSE AND MAINTAINABILITY OF COMMUNICATION SOFTWARE WITH CONVERSATION-AWARE ASPECTS

by

Ali Raza

A dissertation submitted in partial fulfillment
of the requirements for the degree

of

DOCTOR OF PHILOSOPHY

in

Computer Science

Approved:

_____	_____
Dr. Stephen W. Clyde Major Professor	Dr. Curtis Dyreson Committee Member
_____	_____
Dr. Xiaojun Qi Committee Member	Dr. Nicholas Flann Committee Member
_____	_____
Dr. Karl White External Committee Member	Dr. Mark R. McLellan Vice President for Research and Dean of the School of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2014

ABSTRACT

Improving Reuse and Maintainability of Communication

Software with Conversation-aware Aspects

by

Ali Raza, Doctor of Philosophy

Utah State University, 2014

Major Professor: Dr. Stephen W. Clyde
Department: Computer Science

Implementing crosscutting concerns for message-based inter-process communications (IPC) is difficult, even using Aspect-oriented Programming Languages (AOPL) such as AspectJ. Many of these challenges are because the context of communication-related crosscutting concerns is often a conversation consisting of message sends and receives. Current AOPL do not provide pointcuts for weaving of advice into high-level IPC abstractions, like conversations. Other challenges stem from the wide variety of IPC mechanisms, their inherent characteristics, and the many ways in which they can be implemented, even using a common communication framework. This dissertation describes an extension to AspectJ, called CommJ, with which developers can implement communication-related concerns in cohesive and loosely coupled aspects. It also presents preliminary, but encouraging results from a subsequent study that shows seven different ways in which CommJ can improve the reusability and maintainability of applications requiring network communications.

(162 pages)

PUBLIC ABSTRACT

Improving Reuse and Maintainability of Communication Software with Conversation-aware Aspects

Inter-process communications (IPC) are ubiquitous in today's software systems, yet they are rarely treated as first-class programming concepts. Implementing crosscutting concerns for message-based IPC are difficult, even using aspect-oriented programming languages (AOPL) such as AspectJ. Many of these challenges are because the context of a communication-related crosscutting concern is often a conversation consisting of message sends and receives. Hence, developers typically have to implement communication protocols manually using primitive operations, such as connect, send, receive, and close. This dissertation describes an extension to AspectJ, called CommJ, with which developers can implement communication-related concerns in cohesive and loosely coupled aspects. It then presents preliminary, but encouraging results from a subsequent study that begin by defining a reuse and maintenance quality model. Subsequently the results show seven different ways in which CommJ can improve the reusability and maintainability of applications requiring network communications.

Ali Raza

ACKNOWLEDGMENTS

To my father and mother who filled my life with light, love and hope. May God rest their souls in eternal peace.

To Dr. Clyde who guided my steps in this cumulative process. I would like to thank him for his invaluable assistance and support throughout my graduate career.

To Dr. Xiaojun Qi, Dr. Curtis Dyreson, Dr. Nicholar Flann and Dr. Karl White for all their help and suggestions on this dissertation.

Ali Raza

CONTENTS

PAGE

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Inter-process communications *(IPC)* are ubiquitous in today's software systems, yet they are rarely treated as first-class programming concepts. Instead, developers typically have to implement communication protocols manually using primitive operations, such as *connect*, *send*, *receive*, and *close*. For many standard communication protocols, the sequencing and timing of these primitive operations can be relatively complex. For example, consider a distributed system that uses the *Passive File Transfer Protocol* (*Passive FTP*) to move large datasets from a client to a server. In this system, the server would enable communications by listening for connections requests on a published port, usually port 21. A client would then initiate a conversation, i.e., an instance of the *Passive FTP* protocol, via a connect request to the server on that port. The detailed sequences of actions are described in the Figure 1-1.

Neither the client's nor the server's side of the conversation is trivial. In fact, to preserve responsiveness to the multiple simultaneous clients and to end users, both the client and server usually execute parts of the conversation on different threads, making

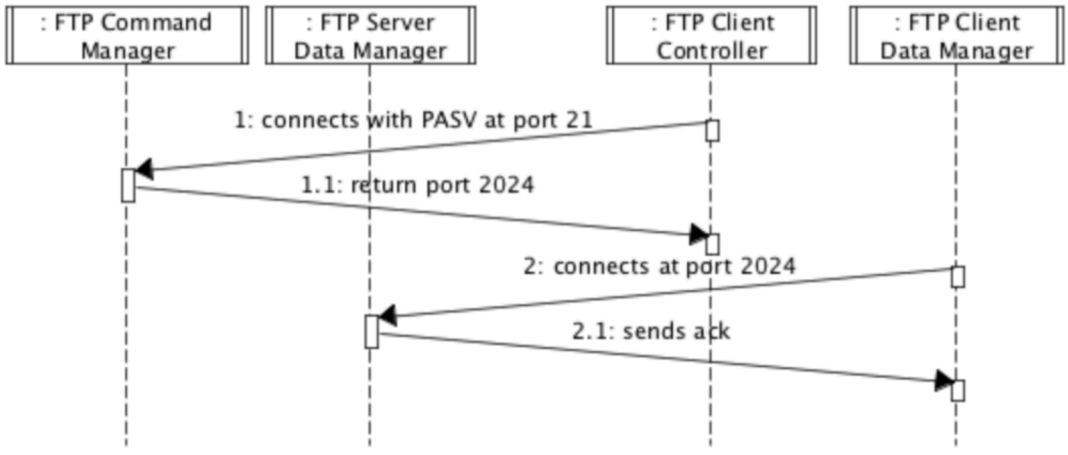

Figure 1-1: *PassiveFTP* Interaction Diagram

them even harder to flow during execution. An *FTP* system could be further complicated by other communication-related requirements, such as:

- logging,
- detecting network or system failures,
- monitoring congestion and
- balancing load across redundant servers.

From a communications perspective, these concerns (and many others not listed above) are what the Aspect-oriented Software Development (*AOSD*) paradigm refers to as *crosscutting concerns*, because they pertain to or *cut through* multiple parts of a core or base system. Directly implementing one or more of these concerns in a typical *FTP* system can cause a *scattering* and *tangling*[1] of code (see Section 6.3 for details).

AOSD, which first started to appear in the literature in 1997 [1], [2], reduces scattering and tangling of code by encapsulating crosscutting concerns in first-class programming constructions, called *aspects* [3]. An aspect is an *Abstract Data Type* (*ADT*), just like classes in strongly typed, class-based object-oriented programming

---

[1] *Scattering* occurs when the same or very similar logic exist in multiple places in the software. *Tangling* occurs when a single software component is complicated by with logic for supporting or secondary concerns. Scattering and tangling often occur together.

languages. However, an aspect can also contain *advice* methods that encapsulate logic for addressing crosscutting concerns and *pointcuts* for describing where and when the advice needs to be executed. A pointcut identifies a set of *joinpoints* – temporal intervals in the execution of the system where and when weaving of advice takes place. Each joinpoint begins and ends relative to static places in the source code, called *shadows* [3].

AspectJ is a programming language that extends to *Java* for aspects, and like many other AOPLs and *Aspect-oriented Frameworks* (*AOF*) [3], [4], [5], [6], it allows programmers to *weave* advice for crosscutting concerns into joinpoints that correspond to constructor calls/executions, methods calls/executions, class attribute references, and exceptions. For a more detailed description of *AspectJ*, see [3].

Since aspects are special *ADT*s that encapsulate certain kinds of design concerns, it is possible for skilled software developers to create reusable object-oriented implementation that do basically the same thing. The real difference between AOP and Object Orientation *OO* is that AOP offers a convenient mechanism for separating crosscutting concerns from core functionality and *obliviousness* [7]. Although poorly named, *obliviousness* is the idea that core functionality should not have to know about crosscutting concerns [8]. Ideally with obliviousness, the crosscutting concerns encapsulated in aspects can be simply added to or removed from a system at build time with no changes to the source code.

The problem is that *AspectJ*, like other AOPs, does not support the weaving of advice into core high-level functional concepts, as does *IPC*. This research extends *AspectJ* so developers can weave crosscutting concern into *IPC* in a modular and reusable way, while keeping the core functionality oblivious to those concerns.

Once we identified the weakness in AOP for weaving advice into IPC, we elaborated on the problem from different dimensions (see Chapter 2) and reviewed the related literature (see Chapter 10). We then pursued the innovation, refinement, and formalization of communication-related joinpoints (see Chapters 3 and 4). This provided a foundation for developing an extension to *AspectJ*, called *CommJ* that allows application programmers to weave aspect behaviors for communication-related crosscutting concerns into such joinpoints. In the next step of our research, we demonstrated the feasibility and utility of *CommJ* by creating a library of reusable communication aspects for common communication-related crosscutting concerns and a suite of non-trivial sample applications that use *CommJ* (see Chapter 5).

Then, we defined an extended quality model, followed up with experiments (Chapters 7 through 9) that investigated the potential implications to the reuse and maintenance to software when developers use *CommJ*. It does so by evaluating certain desirable characteristics through our model (Chapter 6) that can be measured by computable metrics. Based on initial theoretic notions, we hypothesized that developers should see reuse and maintenance improvements relative to seven desired qualities (Chapter 7) defined by the model. Chapter 8 discusses our experiment methodology, which required formal approval from Institutional Review Board (IRB) [9], selection of the sample software application, and identifying interesting crosscutting concerns that would give us good coverage. The methodology also typically included supporting activities such as recruitment and training of the developers. After the experiment, we collected data from the code, surveys, hourly journals, and questionnaires.

From the results (Chapter 10) of the study, we concluded that IPC software components developed with *CommJ* were more cohesive and oblivious. They were also less scattered, and were coupled, complex and smaller in size than similar components programmed in *AspectJ*.

Finally, in Chapter 11 we summarize our research work, contributions and list some avenues for possible future research pursuits. Our first contribution was to define a universe model for communications (UMC) that is rich enough to describe any kind of IPC, supported by the sockets or channels API in a standard JDK. Second, we implemented a library called *CommJ*, including an implementation of UMC that provides the ability to weave advice into program execution before, after, or around complete conversions or individual communication operations. Third, we also developed a reusable aspect library for common communication-related crosscutting concern, which verifies the correctness of UMC. Fourth, we demonstrated the feasibility and utility of *CommJ* and the reusable aspect library through the implementation of application and communication aspects for those applications. Fifth, to measure the effectiveness of *CommJ* in comparison with *AspectJ*, we defined an enhanced version of the Comparison Quality Metrics [10] that measures reusability and maintainability in aspect-oriented programs. Finally, we performed a preliminary experiment to discover whether *CommJ* can help achieve improved reuse and maintainability when a system has involved communication-related crosscutting concerns. These preliminary results lead us to believe that further experimentation with *CommJ* and refinement of its framework could prove to be very beneficial to a wide range of software systems.

CHAPTER 2

BACKGROUND

In general, a skilled programmer can do anything in an *OO* language that could be done in AOP language by making careful design decisions that encapsulate crosscutting concerns in well-modularized classes and hook those features into the base application. To do this, programmers can use a variety of techniques, such as delegates or callbacks, events, the application of a strategy, decorator or template method pattern [11]. However, the developer may end up struggling with code tangling and scattering, unnecessary coupling (i.e., lack of obliviousness), and compromised flexibility. AOP provides a more elegant way of weaving new behaviors into existing code, such that the new functionality is less scattered, tangled, and decoupled from the base application, without compromising functionality.

In AOP, a programmer should only need a *modular reasoning* to discover the code and structure of the crosscutting concerns; whereas she would most likely need *global reasoning* when using traditional OO techniques [12]. Additionally, when using only OO techniques, separating out tangled code from core functionality can cause problems, such as inheritance anomaly [13]. However, in AOP, such tangled code can be refactored and defined into separate aspects as crosscutting concerns. Hence, the attraction of AOP is not that a developer can do more, but that a developer can do some things better, in terms of modularizations with less scattering, less tangling.

2.1. Aspect-oriented Programming Languages, Toolkits, and Framework

Other techniques addressing the same problems emerged at the same time or before aspect-orientation, including monads [14], subject-oriented programming [15], 16], reflection [17, [18], mixins [19], and composition filters [17]. However, the AOP approach seems to have risen to the top as the most influential because it allows better support, better modularity of crosscutting concerns and is consistent with the OO paradigm.

There are different implementations of AOP languages and frameworks, such as *AspectJ* [3], *AspectWorkz* [4], *Spring* AOP [6] and *JBoss* AOP [5]. Though they are semantically similar in terms of their aspect invocation, initialization, access and exception handling routines, their mechanisms differ in programming constructs, syntax, binding, expressiveness (verbosity or compactness), approaches to advise weaving (compile time, load time, or run time), static or dynamic analysis, and their overall acceptance and advancement in academia and industry. Currently, *AspectJ* (now powered by *IBM*) is considered the de facto standard and the most widely used AOP framework for modeling crosscutting concerns due to its *Java*-like structure, powerful expressiveness, and debugging abilities, even though it has some overhead in terms of memory usage and time. In this dissertation, we limit our scope to *AspectJ* for defining the communication-related crosscutting concerns.

2.2. Communications

In general, communications and the mechanisms that implements them, such as channels or sockets, are either connection-oriented or connection-less. Connection-oriented communications require two processes to establish a communication link,

sometimes referred to as a *session*, before exchanging data. This style of communication is very much like a person-to-person telephone call. With connectionless communications, one process can send another process a message without knowing whether that process is ready to receive the message or whether it even exists yet. This style of communication is like traditional postal mail.

We call one or more messages that are logically part of an exchange or collaboration between processes a *conversation*. Conversations can take place with either connection or connectionless communications and can last for just a millisecond or go on for very long periods of time. Like formal interactions between diplomats, electronic conversations between processes follow *protocols* that govern the expected behavior of the participants.

Some protocols are symmetrical, meanings that all participants follow the same rules. However, it is more common for the protocols to be asymmetrical, meaning that each participant acts according to one of several roles. The most common protocols typically consist of two roles: the conversation *initiator* and a *listener*. Sometimes, in the literature, these roles are referred to as *client* and *server*, but these terms often imply other software architectural issues that are not relevant here. Furthermore, it is common for a single process (or even a single thread) in distributed systems to initiate some conversations, while listening for others. So, to avoid confusion with other architectural design choices and focus on the nature of communication, we refer to conversation roles in terms of their essential or distinguishing functions, such as listener, initiator, sender, or *receiver.*

Implementation details can vary with respect to IPC abstractions, but in general their capabilities are similar. One abstraction may provide more flexibility over another in handling a particular situation, but these differences only impact the implementation of the ideas in this dissertation and not the core contributions.

Although IPC abstractions share some common concepts such as listeners, initiators and sessions (see Section 4.1.), they may exhibit various types of well-known communication heterogeneities, such as:

- *Synchronous vs. Asynchronous Communications*: Blocking (sockets) and non-blocking communication (channels) *APIs* in *JDK* are examples of synchronous and asynchronous communications respectively.
- *Unidirectional versus Bi-directional Communications:* Acknowledgment is not required in unidirectional communications but it is either required or inherent in bi-directional communications.
- *Connection-oriented versus Connection-less Communications*: User Datagram Protocol *(UDP)* and Transport Control Protocol *(TCP)* are examples of connection-oriented and connection-less communications respectively.
- *Local versus Global Communications*: Unicast is an example of local communications wherein a broadcast is an example of global communication.
- *Structured versus Unstructured Communications:* Structured style forces objects to send messages to a predefined set of object; however in unstructured communication, an object can exchange messages with any other object.

- *Static versus Dynamic Communications*: With static communications, process identification does not change, whereas with dynamic communications, the process identification may change at run-time.
- *Symmetric versus Asymmetric*: In symmetric communications, the unit or size of message remains fixed but in asymmetric it can vary.

2.3. Crosscutting Concerns in Communication

Despite *AspectJ*'s rich set of pointcut designators, there is still a weakness relative to weaving crosscutting concerns into communication. Specifically, programmers cannot weave aspects into an individual conversation. Since *AspectJ*'s pointcut designators only deal with code constructs, programmers would only be able to weave concerns into the underlying IPC operations, such as connect, send, and receive. Also, the programmers will have to explicitly code mechanisms for tracking individual conversation contexts.

Consider the sample communication-related crosscutting concerns listed in Table 1. If a programmer wants to implement the first one directly in *AspectJ*, he or she would

Table 1. Sample reusable crosscutting concerns in IPC

Aspect Name	Description
TotalTurnAroundTimeMonitor	Provides virtual helper methods for conversations which help programmers to override RAL aspects in their applications
MessageLoggingByConversation	Log messages by conversations in a developer-defined format and repository
MessageEncryption	Add session-level encryption/decryption to communication protocols
NetworkNoiseSimulator	Allows developers to add noise, message log, and message duplication to network communications, which is useful for system testing
NetworkLoadBalancer	Helps programmers balance message loads across two more communication channels
VersionControlAspect	Helps programmers manage multiple version of messages structures for their applications
Authenticator	Tracks consistent and secure multi-step conversations or handing authentication permission in banking domain
QoSTracker	Detects lost, corrupt or out-of-order messages and controlling q

have to implement some advice for the initiating process that would capture the time at which a message was sent and other advice the would capture the time at which the corresponding reply message was received and then compare the two times. However, send and receive logic for the conversation may be in separate modules, may be separated in the execution flow by an undetermined amount of time, and may even be handled on separate execution threads. Furthermore, the initiating process may start many conversations at the same time, and the advice would have to manually correlate the time of a message send with the receive time of the correct corresponding reply. In a nutshell, the weakness of *AspectJ* is that its pointcut designators and joinpoint context are limited to standard programming constructs and do not handle high-level run-time abstractions, like conversation.

To address this problem for communications, we developed an extension to *AspectJ* framework, called *CommJ*, that allows developers to define pointcuts in terms of IPC abstractions and that automatically keeps track of context information for individual conversations. The next chapter provides a high-level overview of *CommJ's* architecture, and Chapter 4 describes its design and implementation.

CHAPTER 3

COMMJ ARCHITECTURE

Figure 3-1 shows an architectural block diagram [20] of *CommJ*, in which the colored blocks represent layers of software or modules, and arrows depict dependencies among these layers. This top-down presentation of the *CommJ* follows a *layered-style architectural design* [21], wherein each layer provides services to the layer above it and uses the services of the layer below it. In general, the *Core CommJ Infrastructure* layer enables communications to be treated as first-class concepts for which developers can define crosscutting concerns in a modular way, i.e., communication aspects. This can help developers manage software complexity while achieving greater reuse and maintainability. Section 3.6 discusses the hoped-for benefits of *CommJ* in more detail, but first Sections 3.1 through 3.5 provide some necessary details about each of the layers, in a top-down order, setting the stage for evaluating whether the hoped-for benefits are achieved.

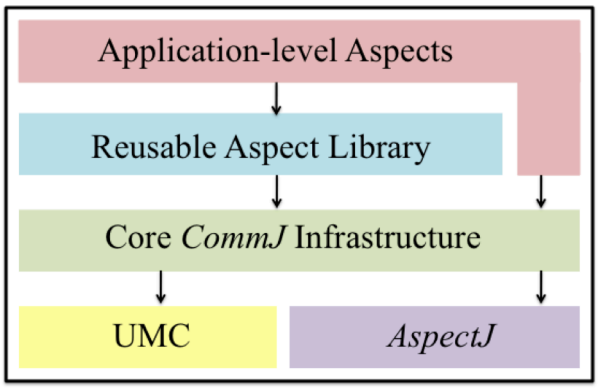

Figure 3-1: *CommJ* Architectural Block Diagram

3.1. Application-level Aspects.

We can write application-level aspects either by using the reusable aspects or base aspects in *CommJ*. For example, an FTP system can have a number of application-level aspects, such as measuring performance, logging, detecting network system failures, load balancing and more. Among them, measuring performance for a multistep crosscutting concern process can be written using the base aspect *MultStepConversation* (Section 4.3.1). We defined each process's role in a multistep conversation using a state machine that describes how the process is expected to act or react with respect to IPC operations. Our aim is for application-level aspects to be easy-to-code, more maintainable and understandable, flexible and modular than similar concerns, programmed in *AspectJ* or a traditional *OO* fashion. (See Chapter 5 for more details).

3.2. Reusable Aspects.

This layer includes a reusable set of *CommJ* aspects that can decrease the development time to program application-level aspects, and help make them more understandable, flexible and oblivious. The reusable aspects are inspired from the key conversation concepts defined in the Universe Model of Communication (UMC). They represent general crosscutting concerns commonly found in applications with significant communication requirements. For example, the *TotalTurnAroundTimeMonitor*, *Authenticator*, and *MessageLoggingByConversation* aspects given in Table 1 are few examples of aspects in the RAL. Section 4.4 describes more about RAL.

3.3. Core *CommJ* Infrastructure.

The *CommJ* Infrastructure is a library that introduces a communication joinpoint model on top of the *AspectJ* joinpoint model, consisting of components for tracking conversation contexts, base aspects that core communication concepts, and a collection of pointcuts for connection and communication operations. Conversation trackers encapsulate hooks into the underlying communications subsystems, e.g., JDK sockets or channels. If those change, one only needs to replace or extend these trackers. The base aspects make use of the context information provided by these conversation trackers and allow RAL or application-level aspects specific to individual conversations. The joinpoints defined in the *CommJ* Infrastructure give the RAL and application-level aspect convenience, reusable pointcuts for most kinds of communications. (See Section 4.1 through 4.3 for more details).

3.4. Universe Model of Communication (UMC).

A universe model for communications (*UMC*) describes a common conceptual understanding about communications, specifically the notation of electronic conversations between multiple processes. In doing so, it models time-sensitive communications-related behavior of execution threads, their processes and the machines (nodes) that host them.

3.4.1. Events

An event can be described as the happening of something. The *UMC* contains three event types: communication event, connection event and exception event. A communication event is the happening of something (related to send or receive) in

message-based communications, at a particular point in time. *Communication Events* are further divided in two types: *Communication Send Event* and *Communication Receive Event*, respectively. The *UMC* states that every receive event must have a corresponding send event. In other words, a send event can exist without a receive event but not conversely. *Communication Events* also exhibit one more special characteristic, namely they can relate to each other. In other words, an event can contain or be associated with many other events. For example, in a distributed application, a thread T_1 can send a message which corresponds to a send event. That message can then trigger a receive message event for some another thread T_2.

Connection Events are happenings related to the setting up of communication channels, and are specialized into four types: *Connect*, *Accept*, *Listen* and *Close* events:

- *Connect Event occurs* when an initiator sends the connect request to a listener
- *Accept Event occurs* when a listener accepts a connect request from an initiator
- *Listen Event occurs* when a listener listens for incoming data
- *Close Event occurs* when a listener or an initiator closes the connection

CommJ does not add any exception events because *AspectJ* already defines a rich set of pointcuts for defining crosscutting concerns that involve exceptions.

The *Thread* class in *UMC* can instantiate and encapsulates multiple send or receives events. A *Communication Event* can be associated with at most one thread. One process can have multiple threads, and a node can host multiple processes. In communication systems, an application may be using multiple nodes, each with several processes. See Figure 3-2.

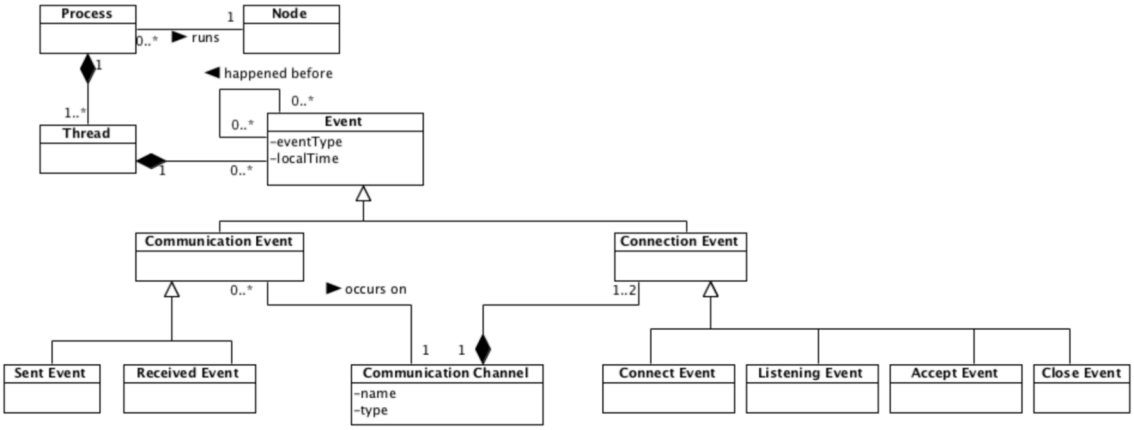

Figure 3-2: UMC for Events

3.4.2. Conversations

In general, a conversation is a sequence of messages that follow communication rules. The UMC generalizes that basic definition to include any sub-sequence. So, a conversation can be:

- an entire conversion from a process's perspective (see A in Figure 3-3)
- any sequence of message send or receive events in the conversation as seen by a process (see B in Figure 3-3)
- a single send or receive event in a conversation (C in Figure 3-3)

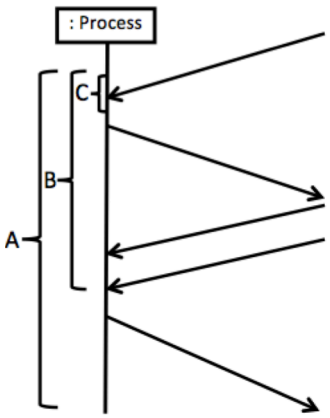

Figure 3-3: Conversations in *UMC*

In Figure 3-4, we see that each conversation in *UMC* can use a set of *Communication Events* on an underlying *Communication Channel*. Any *Communication Event* that happens on a *Communication Channel* is also associated with a particular *Protocol.* A *Conversation* is also capable of keeping track of *Communication Events* that occur in a multithreaded application with multiple channels.

Conversations can also happen during different stages of connection either on initiator or listener in *UMC*. Example of *FTP* (Section 5.4) also elaborates complete connection conversations on both initiator and listener sides respectively.

3.4.3. Channel

Every *Conversation* happens on a *Channe*l (Figure 3-4). A *Channel* also acts as a way of connecting the *Communication Events* with the *Connection Events*. In addition, a *Channel* also abstracts the underlying network-specific components, e.g., Sockets, Channels, etc. into higher-level concepts that are more consistent across platforms.

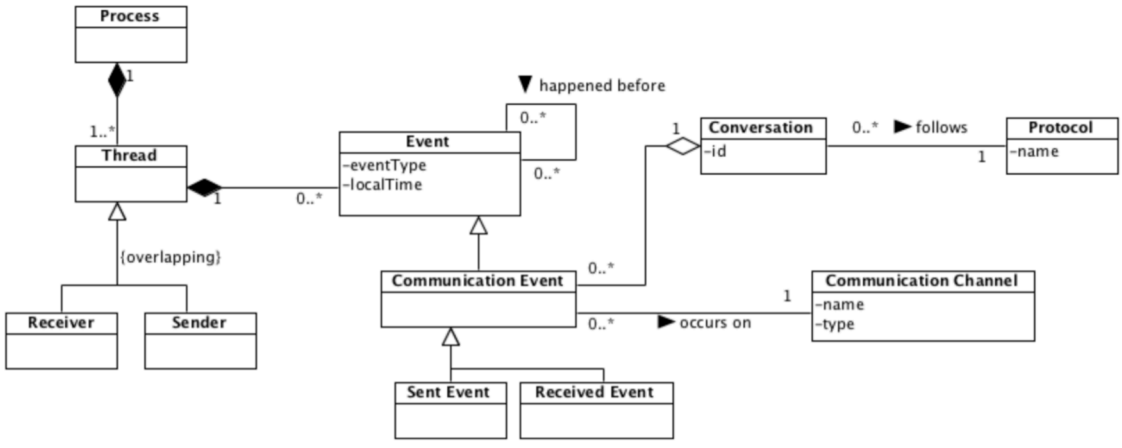

Figure 3-4: UMC for Conversations

3.4.4. Message

A message is a class that encapsulates data exchanged during IPC. Processes or threads in communication systems exchange data through events invocations in *UMC*. *Communication Events* are strongly associated with *Message* instances in the model. Each *Message* can have at most one send and one receive event. Further, *Messages* and *Communication Events* follow similar specialization hierarchies; both are specialized into send and receive types. An instance of *Message* received keeps track of its *ReceivedEvent*, and a *Message* sent knows about its *SentEvent*.

All *CommJ* applications derive their specific message classes from base *Message* class defined by the *UMC* and are implemented in the *CommJ Infrastructure*. The *Message* class realizes a *IMessage* interface that contains method signatures for returning *Message Identifying Information* (*MIF)*. *MIF* may include message identity, message type, conversation identity, protocol specification, and process role, as shown in Figure 3-5. These five elements provide necessary information to identify any message from the registry in *CommJ* and to create and manage various types of conversations.

The *CommJ Infrastructure* dynamically introduces *MIF* in its *initialization aspect*

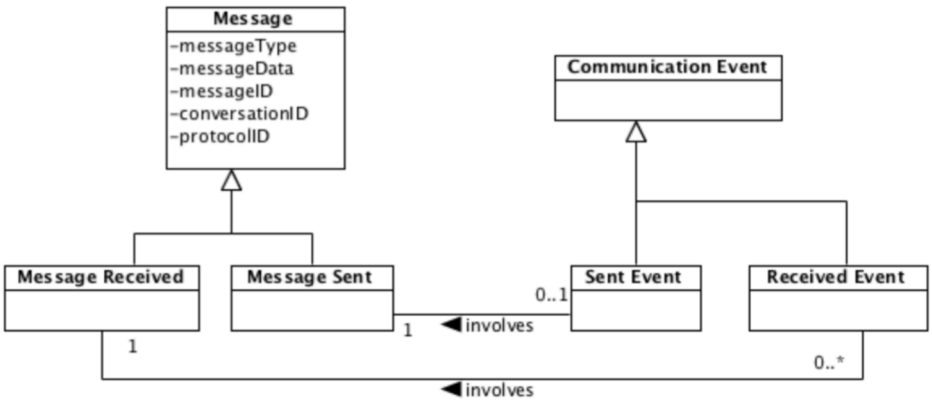

Figure 3-5: UMC for Messages

(Section 4.4). The interface *IMessage* is the only direct dependency between the core application and crosscutting concerns, programmed in *CommJ.*

3.4.5. Connections

A *Process* may be acting in the role of a sender or *Receiver* while handling communication events and as an initiator or a listener while handling *Connection Events*. An initiator can handle only connect and close events whereas a listener can handle *Listen*, *Accept* and *Close* events, respectively. Figure 3-6 illustrates the connection-related concepts in UMC.

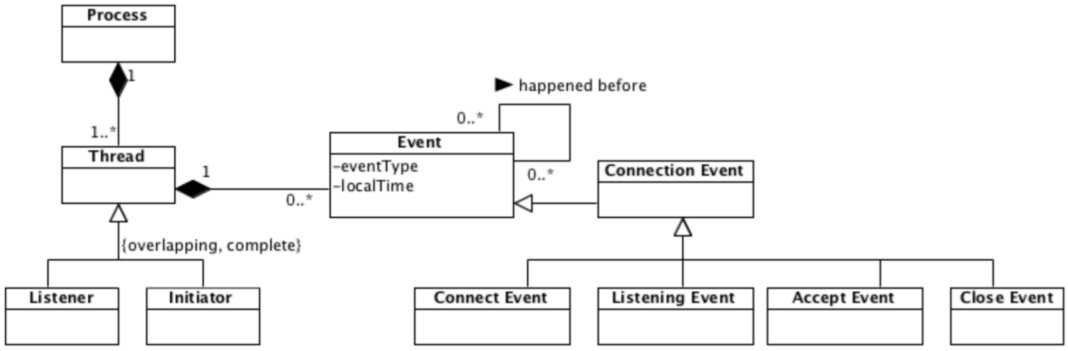

Figure 3-6: UMC for Connections

3.5. *AspectJ*'s Role

The *CommJ* infrastructure realizes the UMC for *AspectJ.* A layer of communication and connection pointcuts in *CommJ* builds on standard *AspectJ* pointcut designators. In addition, the *CommJ Infrastructure* does not constrain the use of any standard *AspectJ* feature, such as programmer-defined pointcuts, advice, inter-type declarations, etc.

3.6. A Design Perspective on *CommJ* with Reference to *AspectJ* and *OOD*

The layers described above can provide software developers with a number of significant benefits when it comes to management the complexity of communications in applications.

3.6.1. Better Abstractions for Communications

Both *AspectJ* and *OOD* weakly encapsulate and modularize *IPC* concerns and would require a multiplicity of pointcut definitions to overcome different types of communication heterogeneities. In comparison, *CommJ* provides better abstractions that unify communication heterogeneities.

3.6.2. Improved Modularity and Obliviousness

In *AspectJ*, writing understandable aspect code for communications is difficult because programming abstractions vary with the underlying communication mechanisms. For example, some communications are connectionless and use datagram packets, while others are connection oriented and use streams. With *CommJ*, developers can program crosscutting concerns in terms of general send, receive, connect, accept and close joinpoints, regardless of the specific communication mechanism or its characteristics. Message data is also uniformly manipulated using a well-defined message interface.

3.6.3. Joinpoint Model Formalizes Communication Joinpoints

AspectJ provides no specific vocabulary for defining communication-related pointcuts. However, in *CommJ*, a developer can define pointcuts using terms that are related directly to IPC concepts.

3.6.4. Better Ways to Detangle Communication Constructs from Core Application.

Java provides various communication abstractions to describe both connection-less and connection-oriented communications. In *CommJ*, a layer of abstraction on top of *AspectJ* helps developers to code aspects in a uniform way, which makes them less tangled, more reusable and more flexible than similar crosscutting concerns, programmed in directly *AspectJ*.

3.6.5. Easy to Code Communication Concerns

It becomes very easy to program communication concerns using pointcuts, such as send, receive, connect, accept and close in *CommJ Infrastructure* with fewer lines of code. In contrast, a developer only using *AspectJ* would need to define considerably more complex pointcuts.

3.6.6. Better Encapsulations and Localized Design Decisions

CommJ provides a rich set of reusable aspects, which localize internal design decisions, and encapsulates many complex mechanisms such as linking of sent messages to received messages. With *AspectJ* only, developers would need complex data structures and explicit mechanisms in order to link these sent and received messages.

3.6.7. Conceptual Model Matches Program Flow Model

In *AspectJ*, the language-to- program *IPC* concerns are different from the program-flow model, but in *CommJ,* due to a library of highly reusable aspects and communication joinpoint model, it matches both conceptual-model and program-flow model of developer.

3.6.8. More Structured Concerns for Communications

In *CommJ*, the application-level code for crosscutting concerns appears to be more elegant and structured than the same concerns programmed in *AspectJ*.

The experience described later in this dissertation provides some preliminary evidence that *CommJ* truly realizes these benefits.

CHAPTER 4

DESIGN AND IMPLEMENTATION

OF A *COMMJ* TOOL SET

Chapter 3 describes the general architecture of *CommJ* along with some fundamental concepts. This chapter discusses the lower-level design and implementation.

4.1. Communication Joinpoints

The UMC serves as a foundation for formalizing communication joinpoints, which fall into two general categories: message-related joint points (Section 2.1) and connection-related joinpoints (Section 2.2), respectively.

4.1.1. Message Event Joinpoints

As mentioned earlier, joinpoints represent places and times where/when advice can be executed. In *AspectJ*, they correspond to constructors, methods, attributes, and exceptions. Advice can be executed before, after, or around these various *contexts*. *CommJ* adds conversations to the list of possible *contexts*, but unlike the contexts in *AspectJ*, a conversation is not tied to a single programming construct but to a conversation. Figure 4-1 represents different kinds of message related joinpoints in *CommJ*.

SendEventJP. It is the region of code, where advice can be woven into, when a communication event related to sending of data, occurs in a process or thread.

ReceiveEventJP. Is the region of code, where advice can be woven into, when a communication event related to receiving of data, occurs in the system.

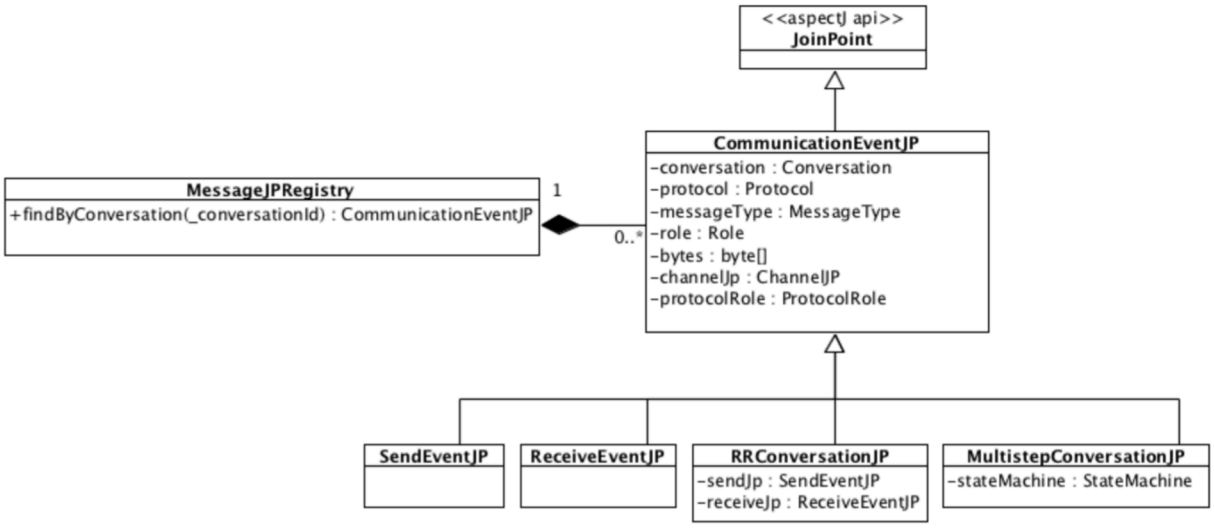

Figure 4-1: Communication Joinpoint and Registry

RequestReplyConversationJP. It represents joinpoints for complete conversations, but they follow basic request-reply protocols. It contains a *SendEventJP* and a *ReceiveEventJP*. *SendEventJP* keeps track of *messageId* whereas the *RecieveEventJP* records a *responseId* for a request-reply type of conversation. An initialization aspect dynamically introduces *MIF* information for all *CommJ* joinpoints. While sending a message, *CommJ* creates an instance of a *SendEventJP* and adds it to the communication registry (which contains communication joint points). Similarly on receiving a message, it creates an instance of a *ReceiveEventJP* and finds a *SendEventJP* from the registry where *messageId* of the former equals *responseId* of the later.

MultiStepConversionJP. It represents joinpoints for entire conversations, as well as joints points for sequences of events. Multiple send and receive events are modeled using a state machine (Section 4.1.5) in a *MultistepConversationJP*.

4.1.2. Registry for Message Joinpoints

When a *MessageJoinPointTracker* discovers a relevant communication event, it creates an instance of a joinpoint class, e.g., *SendEventJP*, correlates it with other events in the same conversation, and then adds it to a registry, namely, the *MessageJPRegistry* shown in Figure 4-2. Any communication aspect can access these joinpoint objects to obtain context information, like the conversation's start time, channel, or the protocol.

4.1.3. Connection Joinpoints

As mentioned earlier, a connection can contain a sequence of *Connect, Accept, Listen*, and *Close* events. Connection joinpoints in *CommJ* are divided in two categories, i.e., joinpoints for initiator and listener respectively (See Figure 4-2 for more details).

ConnectJP. Initiator creates a *ConnectJP*. It encapsulates the connection information related to underlying sockets and channels along with their local and remote addresses.

AcceptJP. Listener creates an *AcceptJP* on receiving a connection request from the initiator.

ChannelJP. It acts like a bridge between communication joinpoints and connection joinpoints.

CloseJP. Both initiator and listener need to instantiate this joinpoint. It encapsulates the closing of connection for an underlying socket or a channel. A listener *AcceptJP* and initiator *ConnectJP* maintains an association with *CloseJP* using a *ChannelJP*.

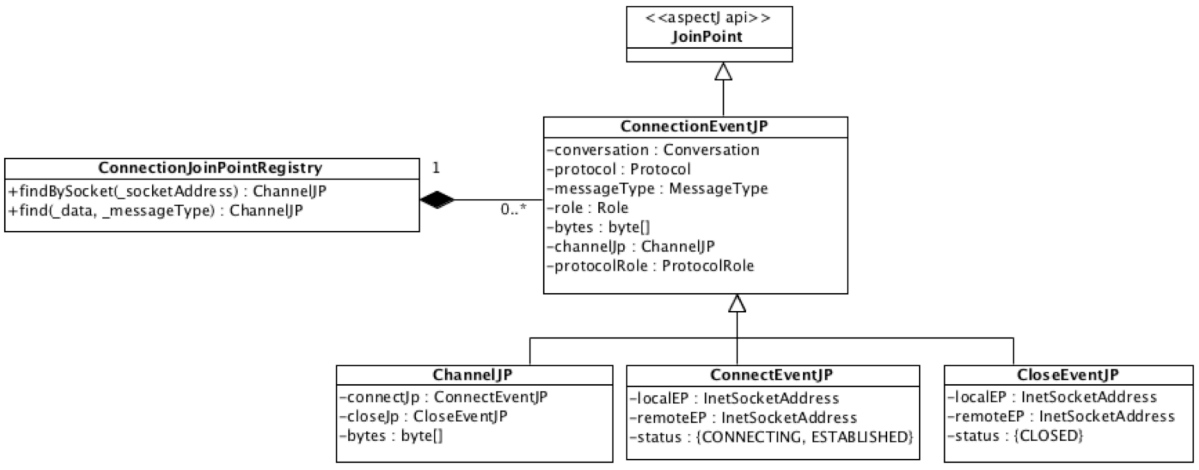

Figure 4-2: Connection Joinpoint and Registry

4.1.4. Registry for Connection Joinpoints

When an *InitiatorJoinPointTracker* or a *ListenerJoinPointTracker* discovers a relevant connection event, it creates an instance of a joinpoint class, e.g., *ConnectJP*, *AcceptJP, ChannelJP or CloseJP*; further it correlates with other events in the same connection-related conversation, and then adds it to a registry, namely the *ConnectionJPRegistry* shown in Figure 4-2. Any connection-related aspect can access these joinpoint objects to obtain context information, such as the connection underlying socket or channel information, connection state or connection start time.

4.2. Joinpoint Trackers

Behind the scenes, *CommJ* uses *JoinpointTrackers*, which are monitors [22] that perform pattern matching on communication events and connection events to track individual events and to organize them into high-level conversation contexts. Since the monitoring of communications is itself a crosscutting concern, *JoinpointTrackers* are implemented as aspects that weave the necessary monitoring logic into places where a

communication event may take place. In *CommJ*, there can be two types of event trackers, i.e., message joinpoint tracker and connection joinpoint tracker, respectively.

4.2.1. Message Joinpoint Tracker

The Message Event Tracker (Figure 4-3) in *CommJ* crosscuts the send and receive events for both reliable and unreliable communication in the core application and defines a set of pointcuts in the simple send and receive abstractions. In *CommJ, MessageJoinpointTracker* is an aspect that hides communication related abstractions in the core application.

This aspect defines pointcuts in the send and receive abstractions (Figure 4-4) by overcoming the syntactic and semantic variations, defined in *Java* pre-built sockets and channels libraries. It provides simple and elegant communication pointcuts, which are rich enough to encapsulate abstractions for both connection-oriented and connectionless protocols. Hence, *MessageJoinpointTracker* creates two clean, well-encapsulated

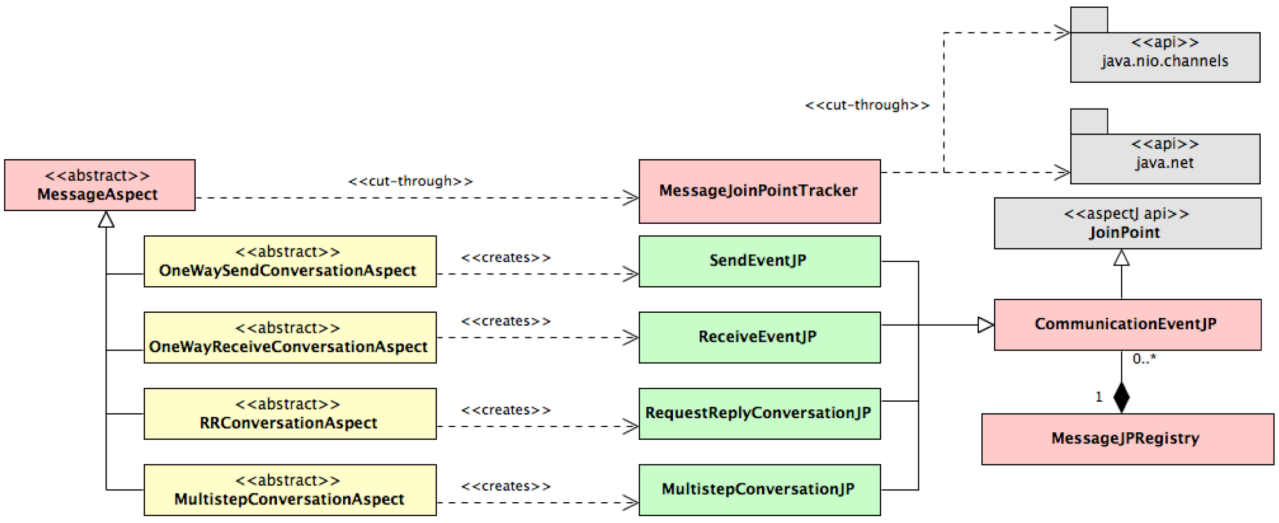

Figure 4-3: *CommJ* Message Event Join Points and Reusable Aspects

communications related abstractions for all types of *read* and *write operations*.

- *Communication pointcuts for reads:* These pointcuts unify syntactic and semantic variations in *Java* communication libraries and crosscut sockets and channels *read* operations.
- *Communication pointcuts for writes:* These pointcuts unify syntactic and semantic variations in *Java* communication libraries and crosscut sockets and channels *write* operations.

```
public aspect MessageJoinPointTracker {

 private pointcut SocketRead(Socket _socket, byte[] _buffer, int _len) :
        call(* Socket+.read(byte[], ..)) && target(_socket) && args(_buffer, _len);

 private pointcut ChannelRead(SocketChannel _channel, ByteBuffer _buffer) :
        call(* SocketChannel+.read(ByteBuffer)) && target(_channel) && args(_buffer) ||
        call(* DatagramChannel+.receive(ByteBuffer)) && target(_channel) && args(_buffer) ;

 public pointcut SocketWrite(Socket _socket, byte[] _data, int _length) :
        call(void Socket+.write(byte[], int)) && target(_socket) && args(_data, _length);

 public pointcut ChannelWrite(SocketChannel _channel, ByteBuffer _data) :
        call(* SocketChannel+.write(ByteBuffer)) && target(_channel) && args(_data);

 public pointcut DatagramChannelWrite(DatagramChannel _channel, ByteBuffer _data, SocketAddress _addr) :
        call(* DatagramChannel+.send(ByteBuffer, SocketAddress)) && target(_channel) && args(_data, _addr)
;

 private pointcut DatagramChannelRead(DatagramChannel _channel, ByteBuffer _buffer) :
        call(* DatagramChannel+.receive(ByteBuffer)) && target(_channel) && args(_buffer);
  …
}
```

Figure 4-4: A Code Snippet of *MessageJointPointTracker*

4.2.2. Connection Joinpoint Trackers

Connection Joinpoint trackers are categorized into *Initiator Joinpoint Tracker* and *Listener Joinpoint Tracker*, respectively. They crosscut the syntactic and semantic variations, exist in both reliable and unreliable communications and unify them into a set of pointcuts in the abstractions of *channel, connect*, *accept* and *close*, respectively.

Listener Joinpoint Tracker. It defines two simple pointcuts, which manages all connection-related abstractions and styles related to the *listener* for connectionless and connection-oriented communications. It encapsulates *AcceptJP*, *CloseJP* and *ChannelJP* (Section 4.2). Figure 4-5 describes the general architecture about the Listener joinpoint Tracker, and Figure 4-6 presents its code snippets.

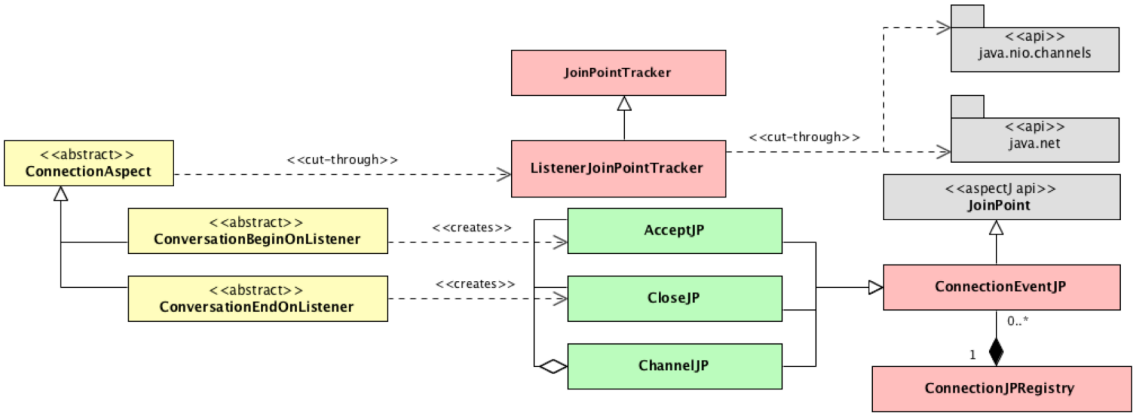

Figure 4-5: Listener Joinpoint and Base Aspects

```
public aspect ListenerJoinPointTracker {
 private pointcut SocketAccept(Socket _socket, InetSocketAddress _remoteEP):
        call(* Socket+.accept(..)) && target(_socket) && args(_remoteEP);

 pointcut ChannelAccept(ServerSocketChannel _serverSocketChannel) :
        call(* ServerSocketChannel+.accept()) && target(_serverSocketChannel) ;


 pointcut ChannelClose(ServerSocketChannel _serverSocketChannel) :
        call(* ServerSocketChannel.close()) && target(_serverSocketChannel);

   .........

}
```

Figure 4-6: A Code Snippet of *ListenerJoinPointTracker*

- *Accept pointcut:* It crosscuts the accept operation for sockets and channels in *Java API* while trying to establish a connection request from the initiator.
- *Close pointcut:* It crosscuts close operation for sockets and channels in *Java* API while closing connection on the listener.

Initiator Joinpoint Tracker. The *InitiatorJoinPointTracker* defines three pointcuts, which manage all connection-related abstractions for an Initiator in both connectionless and connection-oriented communications. It encapsulates *ConnectJP*, *CloseJP* and *ChannelJP* (Section 4.2). Figure 4-7 describes the general architecture about the Initiator joinpoint Tracker and Figure 4-8 presents its code snippets.

- *Connect pointcut:* It is a crosscut connect operation for sockets and channels in the *Java API* on the initiator side while requesting the listener to establish a connection. Additionally, *Connect finish pointcut* defines the finished operation on the initiator side when the listener has successfully established a connection.

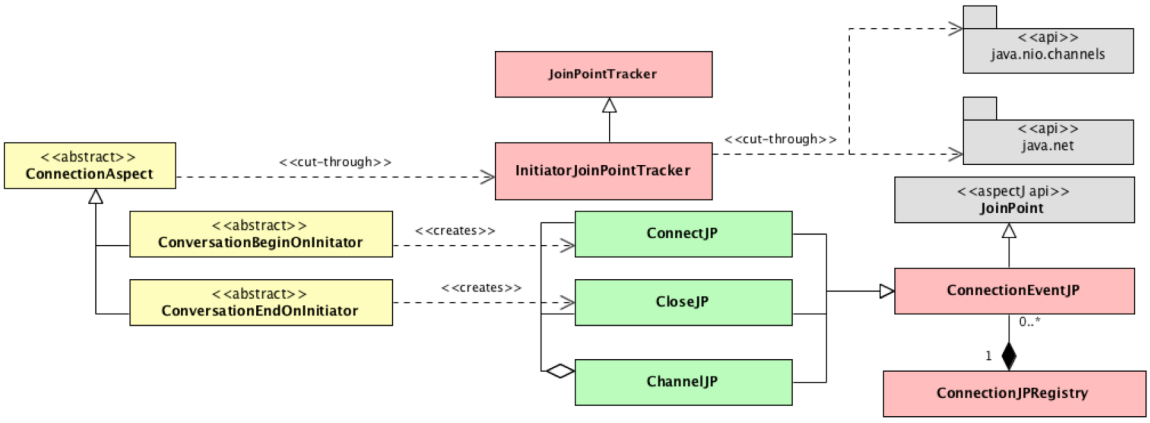

Figure 4-7: Connection Joinpoint and Base Aspects

```
public aspect InitiatorJoinPointTracker{
 private pointcut SocketConnectStyle1():
        call(Socket.new());

 private pointcut SocketConnectStyle2(InetAddress _address, int _port):
        call(Socket+.new(InetAddress, int)) && args(_address, _port);

 ……

 pointcut ChannelConnect(SocketChannel _socketChannel, InetSocketAddress _remoteEP) :
        call(* SocketChannel.connect(..)) && target(_socketChannel) && args(_remoteEP);

 pointcut ChannelConnectFinish(SocketChannel _socketChannel) :
        call(* SocketChannel+.finishConnect(..)) && target(_socketChannel);

 private pointcut SocketClose(Socket _socket):
        call(* Socket+.close(..)) && target(_socket);

 pointcut ClientChannelClose(SocketChannel _channel) :
        call(* SocketChannel.close()) && target(_channel);

 ……

}
```

Figure 4-8: A Code Snippet of InitiatorJoinPointTracker

- *Close pointcut:* This pointcut defines close operation on initiator side Base Aspects.

The *CommJ Infrastructure* contains two kinds of base aspects, *Communication* aspects and *Connection* aspects. They cut through their respective joinpoint trackers and provide pointcuts in the abstractions of high-level *IPC* methods.

4.3. Base Aspects

CommJ implements communication-related crosscutting concerns as aspects, derived from base conversation aspects (described below) using communication joinpoint trackers.

4.3.1. MessageAspect

All communication aspects are ultimately derived from the abstract *MessageAspect* class, which provides concrete pointcuts that dynamically track send and

receive events. See Figure 4-9. It is important to note that these pointcuts take joinpoint objects as parameters, because this is how advice woven into these pointcuts, can access conversation contexts.

The four specializations of *MessageAspect* correspond to four different kinds of conversation contexts, as mentioned earlier, and extend *MessageAspect* with pointcut abstractions that are meaningful to those contexts. Developers can create their own application-level communication aspects that inherit from these aspects and include their own advice based on these pointcuts.

One-way send (OWS). An *OWS* conversation involves only one send event on the initiator's side. For the initiator, the conversation automatically ends after send event is finished (See Figure 4-10).One *way receive (OWR).* An *OWR* conversation for a listener involves only one receive event. The conversation automatically ends for the listener after a receive event (see Figure 4-11).

```
public abstract aspect MessageAspect{
        public pointcut MessageSend(SendEventJP _sendJp) ....
        public pointcut MessageRecieve(ReceiveEventJP _receiveJp) ....
}
```

Figure 4-9: A Code Snippet of Message Aspect

```
public abstract aspect OneWaySendAspect  extends MessageAspect{
        public pointcut ConversationBegin(SendEventJP _sendEventJp)....
        void around(SendEventJP _sendJp) : MessageSend(_sendJp){

                ....
}
```

Figure 4-10: A Code Snippet of OneWaySendAspect

```
public abstract aspect OneWayReceiveAspect extends MessageAspect{
        public pointcut ConversationEnd(ReceiveEventJP _receiveEventJp)....
        void around(ReceiveEventJP _receiveJp) : MessageRecieve(_receiveJp){

                ....
        }
        ....
}
```

Figure 4-11: A Code Snippet of OneWayReceiveAspect

```
public abstract aspect RRConversationAspect extends MessageAspect{
        public pointcut ConversationBegin(RequestReplyConversationJP _requestReplyJp) ....
        public pointcut ConversationEnd(RequestReplyConversationJP _requestReplyJp) ....
        ....
}
```

Figure 4-12: A Code Snippet of RRConversationAspect

Bi-directional (Request/Reply style of Conversation). Bi-directional conversations require a successful round-trip of a send and receive events. An *RRConversationAspect*, which applies to bi-directional conversations, defines pointcuts *StartConversation* and *EndConversation*. The *StartConversation* creates a *RequestReplyConversationJP* and starts a conversation when a sender invokes a sent event, the *EndConversation* retrieves the matching *RequestReplyConversationJP* from the *MessageJPRegistry* and ends a conversation when a *Receiver* invokes a receive event (See Figure 4-12 for more details).

Multi-step Conversations. Multi-step conversation involves any combination of send and receive events without any specific order. For example, few variations in multi-step conversations are as follows: one send event and multiple receive events; multiple send events and one receive event; multiple send events and multiple receive events or any complex model of send and receive events.

```
abstract aspect MultistepConversationAspect extends MessageAspect{
  public pointcut ConversationBegin(MultistepConversationJP _multiStepJp)....
  public pointcut ConversationEnd(MultistepConversationJP _multiStepJp)....

  void around(SendEventJP _sendJp) : MessageSend(_sendJp){
        ....
  }

  void around(ReceiveEventJP _receiveJp) : MessageRecieve(_receiveJp){

        ....
  }
        ....
}
```

Figure 4-13: A Code Snippet of MultistepConversationAspect

We programmed the multistep conversation aspect in Figure 4-13 by deriving from *MessageAspect* class and thereby inheriting the *MessageSend* and *MessageReceive* pointcuts. A multistep conversation retrieves message, role, protocol and conversation information from *Message* class and creates a state machine instance if it doesn't already exist. During one application session, an aspect may apply several concurrent conversations for one type of state machine (protocol). The context for each conversation is maintained in terms of its own current state and association state machine instance. (See Figure 4-14 for more details on the state machines).

CommJ State Machine for Multistep Conversations. In general, there are two types of state machines. Mealy and Moor state machines [18]. Mealy state machine is a finite state machine whose output values are determined both by its current state and the current inputs whereas in the Moore state machine, the output values are determined solely by its current state. Mealy state machines are better suited for CommJ because they can be defined in terms of transitions triggers, which correspond to message events and message types. The design of the state machine for multistep conversation is shown in Figure 4-14 and code snippet is in Figure 4-15. A *CommJ* state machine has the following components: *State* and *Transition*. A *State* encapsulates the state name, whether it is in

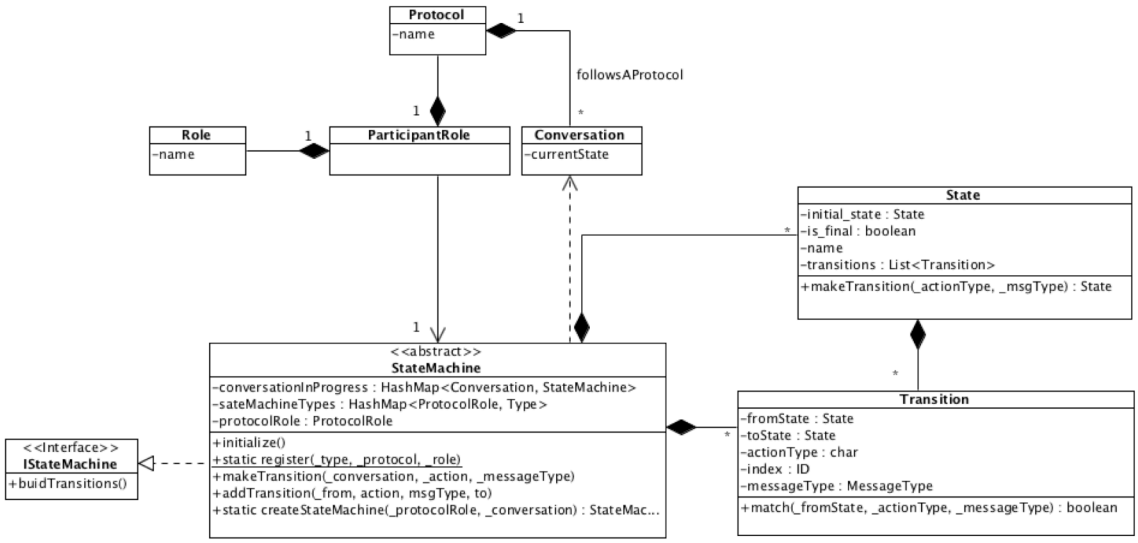

Figure 4-14: Design of Multi-step State Machine

initial or final state, and its list of transitions. *Transition* is defined using four basic elements: *ActionType*, *MessageType*, *FromState*, and *ToState*. The *ActionType* is transition trigger and can be either a send or receive action. The *MessageType* is a filter or guard that specifies what types of messages may trigger the transition. *FromState* defines the state before transition and *ToState* defines the target state after transition.

ConversationInProgress. A distributed application may be communicating with multiple other processes, which are also involved in a multi-step conversation. A state machine instance can keeps track of these multiple concurrent conversations by maintaining a collection of in-progress conversations.

StateMachineTypes. When an application is loaded in memory, all types of application-level state machine classes are initialized and stored in *StateMachineTypes* - a hash map type of data structure. This hash map keeps a mapping between application classes and state machine types. *Register()* method of the abstract state machine in *CommJ* is called when applications are loaded through *static block initialization* (Figure 4-15).

4.3.2. Connection Aspects

A *Connection Aspect* derives from a *CommJ* base aspect, which crosscuts *ListenerJoinPoinTracker* and *InitiatorJoinPointTracker pointcuts*. The base connection aspect defines the following four pointcuts (See Figure 4-15):

Connect pointcut. It crosscuts *InitiatorJoinPointTracker* connection related pointcut and provides Connect pointcut.

Accept pointcut. It crosscuts *ListenerJoinPointTracker* accept related pointcuts and provides *Accept* pointcut.

CloseServer pointcut. It crosscuts *ListenerJoinPointTracker* "close connection" pointcuts and provides *Close* pointcut.

CloseClient pointcut. It crosscuts *InitiatorJoinPointTracker* "close connection" pointcuts and provides *Close* pointcut.

4.3.3. Complete Connection Conversation.

The complete *Connection Conversation* aspect is inherited from *ConnectionAspect* (Figure 4-16) and defines pointcuts that help programmers to define

```
public abstract aspect ConnectionAspect {

 public pointcut Connect(ConnectEventJP _connectJp) :
       within(InitiatorJoinPointTracker) &&
             execution(* InitiatorJoinPointTracker.ChannelConnect(..)) && args(_connectJp);

 public pointcut Accept(ConnectEventJP _connectJp) :
       within(ListenerJoinPointTracker) &&
             execution(void ListenerJoinPointTracker.ChannelAccept(..)) && args(_connectJp);

 public pointcut CloseServer(CloseEventJP _closeJp) :
       within(ListenerJoinPointTracker) &&
            execution(void ListenerJoinPointTracker.CloseServerEventJointPoint(..)) &&
          args(_closeJp);

 public pointcut CloseClient(CloseEventJP _closeJp) : within(InitiatorJoinPointTracker) &&
       execution(void InitiatorJoinPointTracker.CloseClientEventJointPoint(..)) &&
          args(_closeJp);
}
```

Figure 4-15: A Code Snippet of Connection Aspect

```
public  aspect CompleteConnectionAspect extends ConnectionAspect{

 public pointcut ConversationBeginOnInitiator(ChannelJP _channelJp) :
       execution(* CompleteConnectionAspect.BeginOnInitiator(ChannelJP)) && args(_channelJp);

 public pointcut ConversationBeginOnListener(ChannelJP _channelJp) :
       execution(* CompleteConnectionAspect.BeginOnListner(ChannelJP)) && args(_channelJp);

 public pointcut ConversationEndOnListener(ChannelJP _channelJp) :
       execution(* CompleteConnectionAspect.EndListener(ChannelJP)) &&  args(_channelJp);

 public pointcut ConversationEndOnInitiator(ChannelJP _channelJp) :
       execution(* CompleteConnectionAspect.EndInitiator(ChannelJP)) && args(_channelJp);

}
```

Figure 4-16: A Code Snippet of Complete Connection Aspect

conversations for total connection time on both listener as well as on the initiator sides.

CompleteConnectionAspect (Figure 4-16) is a reusable connection related conversation aspect. It extends from *ConnectionAspect* and provides following pointcuts:

- *ConversationBeginOnInitiator.* This pointcut crosscuts the state of request to establish a connection on initiator side and marks it as start of the Initiator connection conversation
- *ConversationEndOnInitiator.* This pointcut crosscuts the closing connection on initiator side and marks it as end of the initiator connection conversation
- *ConversationBeginOnListener.* This pointcut marks the start of connection related conversation when Listener tries to accept a connection request.
- *ConversationEndOnListener.* This pointcut marks the end of connection related conversation when Listener tries to close a connection

```
public  abstract aspect Initialization {

  public static Conversation conversation = new Conversation();
  public static Protocol protocol = null;
  public static Role role = null;

  private pointcut ConfigureMessage(IMessage _message) :
        execution(void utilities.IMessage.setMessage(..)) && target(_message);
  public  pointcut ConfigureProtocolRole() :execution(void *.main(..));
  public static HashMap<Class<?>, Class<?>> mappings = new HashMap<Class<?>, Class<?>>();

  public abstract void defineMappng();

  before():ConfigureProtocolRole(){
        defineMappng();
         Class<?> className = thisJoinPointStaticPart.getSignature().getDeclaringType();
         invokeRoleAndProtocol(className);
   }


  after(IMessage _message):ConfigureMessage(_message){
        _message.setProtocol(protocol);
        _message.setRole(role);
        _message.setConversation(conversation);
   }

  public void addMapping(Class<?> _classA, Class<?> _classB){
        mappings.put(_classA, _classB);
   }

}
```

Figure 4-17: A Code Snippet of *CommJ Initialization Aspect*

4.3.4. CommJ Initialization Aspect

This aspect (Figure 4-17) loads application specific state machines when communication process starts. Besides initialization of state machines, this aspect also crosscut initialization of messages and introduces conversation, role, protocol and message identity information before sending or after receiving these messages.

4.4. Reusable Aspects Library (RAL)

Aspects in the RAL are also derived from the base aspects in *CommJ*. They represent general crosscutting concerns commonly found in applications with significant communication requirements. Table 1 lists some of the aspects currently in the RAL and Figure 4-18 shows part of the implementation of first one, *TotalTurnAroundTime-*

Monitor. Note how the advice in this aspect follows the *Template Method* pattern [8]. This allows developers to quickly adapt it to the specific needs of their application by overriding the *Begin* and *End* methods. Other aspects in the RAL make use of this and other reuse techniques so developer can easily integrate them into existing or new applications. We expect that RAL will continue to grow as new generally applicable communication aspects are discovered, implemented, and documented.

4.4.1. Turn-around Time Aspect in RAL

As mentioned, aspect developers implement and add application-level aspects into core application logic by either reusing RAL aspects or specializing the base aspects in *CommJ*. As an example, this section describes the implementation of an application-level aspect that weaves performance measurements in the multistep protocol, introduced in the previous section. For discussion purposes, assume that the performance measurements are a rolling window of throughput and average-conversation turn-around time statistics. Also, assume that the core application considers a unit of work to be the completion of a conversation that follows this protocol. So, we can measure throughput for a unit of time, say 1 minute, by simply counting the number of these conversations completed in that minute. The average turn-around time is the average of timespans from conversation start times to conversation end times. The rolling window keeps track of these statistics for the current minute and 10 previous minutes.

First notice how this advice is derived from *TotalTurnAroundTimeAspect* and in doing so, it can reuse its implementation of the conversation turnaround time concept directly. Then, it adds the *Stats* array for holding the rolling window of statistics and some additional behavior to the ending of a conversation to compute the statistics.

```
public aspect TotalTurnAroundTimeMonitor extends MultistepConversationAspect{
    private long startTime = 0;
    private long turnAroundTime = 0;
    before(MultistepConversationJP jp): ConversationBegin(jp){
        startTime = System.currentTimeMillis();
        Begin(jp);
    }
    after(MultistepConversationJP jp): ConversationEnd(jp){
        long turnaroundTime = (System.currentTimeMillis() –
                startTime)/1000;
        End(multiStepJP);
    }
    public getTurnAroundTime { return turnAroundTime; }
   protected void Begin(MultistepConversationJP jp){
      // Specialization of this aspect should override the method
   }
   protected void End(MultistepConversationJP jp){
      // Specialization of this aspect should override the method
   }
   …
}
```

Figure 4-18: A Code Snippet of *TurnAroundTimeMonitor*

CHAPTER 5

APPLICATION-LEVEL ASPECTS

As mentioned, aspect developers implement and add application-level aspects into core application logic by either reusing RAL aspects or specializing the base aspects in *CommJ*. This chapter provides four examples of communication and connection related crosscutting concerns implemented with *CommJ*.

5.1. Measuring Performance in Multistep Conversation Processes

This example discusses the design and implementation of measuring the total turnaround time for a multistep conversation. Consider a communication protocol involving three processes, *A*, *B*, and *C*, wherein A starts a conversation by sending a message to *B* and waits for a response. When *A* receives a response *B*, it sends a message

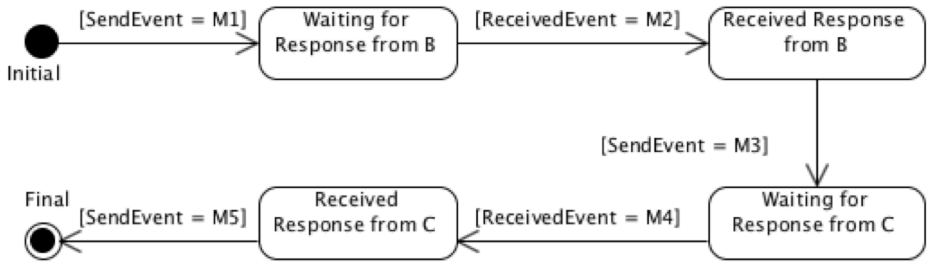

Figure 5-1: State Machine for the *A ProcessRole*

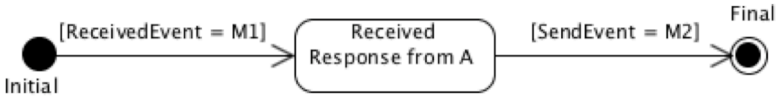

Figure 5-2: State Machine for the *B ProcessRole*

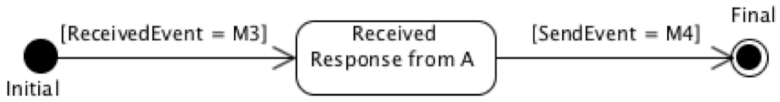

Figure 5-3: State Machine for the *C ProcessRole*

```
public class ProcessRoleA extends StateMachine{
  ....
 @Override
 public void buildTransitions(){
  addTransition("Initial", 'S', "M1", "WaitingRspFromB");
  addTransition("WaitingRspFromB ", 'R', "M2", " ReceivedRspFromB");
  addTransition("ReceivedRspFromB", 'S', "M3", " WaitingRspFromC");
  addTransition("WaitingRspFromC",  'R', "M4"," ReceivedRspFromC");
  addTransition("ReceivedRspFromC",  'S', "M5"," Final");
 }
        ....
}
```

Figure 5-4: State Machine Configuration for ProcessRoleA

to *C* and waits for a response. When *A* receives a response from *C*, it sends a final message to both *B* and *C*. Figure 5-1 shows a finite state machine for the *A ProcessRole* of this protocol. The behaviors for *B* and *C ProcessRoles* are considerably simpler and are shown in Figures 5-2 and 5-3, respectively.

5.1.1. Design and Implementation

The *CommJ StateMachine* class includes a *buildTransitions* method that allows developers to define state machines in terms of states and message-event transitions. Figure 5-4 shows the implementation of this method to define a *StateMachine* for the *A ProcessRole*.

For discussion purposes, assume that the performance measurements are a rolling window of throughput and average-conversation turn-around time statistics. Also, assume that the core application considers a unit of work to be the completion of a conversation that follows this protocol. So, throughput can be measured for a unit of time, say 1 minute, by simply counting the number of these conversations completed in 1 minute. The average turn-around time is the average of timespans from conversation start times to conversations end time. The rolling window keeps track of these statistics for the current minute and the 10 previous minutes. Figure 5-5 shows the key pieces of code for an aspect that implement this performance measure crosscutting concern.

First notice how the aspect is derived from *TotalTurnAroundTimeAspect* and in doing so, it can reuse its implementation of the conversation turnaround time concept directly. Then, it adds the *Stats* array for holding the rolling window of statistics and some additional behavior to the ending of a conversation to compute the statistics.

5.2. Version Control Aspect

This example discusses the design and implementation of an aspect that can coordinate communications when different processes are following different version of a protocol. Imagine that the protocol discussed in the previous example has evolved over time, resulting in multiple versions of the messages' syntax. If A process is following the updated syntax rules and trying to communicate with B or C processes that are following rules from prior versions, there will be communication errors. Ideally, it would be nice to allow seamless independent upgrading to any of the processes without effecting the communications.

```
public aspect MyAppPerformanceMonitor
          extends TotalTurnAroundTimeMonitor{

  private Stats[] statsList = new ArrayList[11];
  private int currentStatsIndex = 0;

  @Override
  protected void End(MultistepConversationJP jp) {
   // Get number of elapsed minutes since beginning of current Stats
   long elapsedMinutes = Min(Stats[currentStatsIndex]. getMinutesSinceStartTime(), 10);
   // Roll Stats window forward, if necessary
   for (int i=0; i<elapsedMinutes; i++){
     currentStatsIndex++;
     if (currentStatsIndex>10)
        currentStatsIndex=0;
        Stats[currentStatsIndex].Reset();
   }
   currentStats.addCompleteConversation(getTurnaroundTime);
  }
}

class Stats{
  private long startTime;
  private int completeConvCount;
  private double avgTurnaroundTime;

  public Stats{
    Reset();
  }

  public Reset(){
    startTime = currentTime;
    completeConvCount = 0;
    avgTurnaroundTime = 0;
  }
  public long getMinutesSinceStartTime() {
    // using current time, compute and return the number of minutes since the start time of this Stats
object. A zero means we still in the same minute
  }

  public void addCompleteConversation(double newTurnaroundTime) {
    avgTurnaroundTime = ((completeConvCount*avgTurnaroundTime) +
newTurnaroundTime)/(++completedConvCount);
  }
}
```

Figure 5-5: Performance Measure Crosscutting Concern

5.2.1. Design and Implementation

The application-level version control aspects in Figures 5-6 and 5-7 extend RAL aspects discussed Section 4.5. On sending the messages, *OneWaySendAspect* ensures that it is sending the most recent version of messages. Similarly, on receiving the messages, *OneWayReceiveAspect* verifies that received message is also in the most recent version.

```
public aspect SendVersionControlAspect  extends OneWaySendAspect{
    ....
    void around(SendEventJP _sendEventJp): ConversationBegin(_sendEventJp){
        //code that check and update the most recent version of messages being sent
    }
}
```

Figure 5-6: Version Control Aspect for Messages Sent

```
public aspect ReceivedVersionControlAspect  extends OneWayReceiveAspect{
    ....
    void around(ReceiveEventJP _receiveEventJp): ConversationEnd(_receiveEventJp){
        //code that check and update the most recent version of received messages
    }
}
```

Figure 5-7: Version Control Aspect for Messages Received

5.3. Managing Quality of Service in Weather Station Data Collection

This example discusses the design and implementation of Quality of Service (QoS) control aspect in the context of a system that collects data from weather stations, referred to here as a WSDC. The QoS control involves managing the compression level for data transmitted by collection nodes in the WSDC. The aspect manages the QoS through a separate QoS channel that monitors and adjusts the compression level between *Transmitter* and *Receiver.*

Typically a weather station is a facility either on land or sea, with instruments and equipment for observing atmospheric conditions to provide information for weather forecasts and to study the weather and climate. The measurements are usually taken including temperature, barometric pressure, wind speed, wind direcFtion and precipitation amounts. Observations can be taken manually or automatically and at regular intervals. Weather conditions out at sea are taken by ships and buoys that measure slightly different metrological quantities such as sea surface temperature, wave height, and wave period [23].

Following are the important devices for getting the data at a typical Weather Station:

- *Thermometer* for measuring temperature
- *Anemometer* for measuring wind speed
- *Wind vane* for measuring wind direction
- *Hygrometer* for measuring humidity
- *Barometer* for measuring atmospheric pressure
- *Ceilometer* for measure cloud height
- Present weather sensor or *visibility sensor*
- *Rain gauge* for measuring liquid-equivalent precipitation
- *Ultrasonic snow depth sensor* for measuring depth of snow
- *Pyranometer* for measuring solar radiations

The standard mast heights used with typical weather stations are 2, 3, 10 and 30 meters, respectively. These sizes are used as standards for differing applications.

- The 2-meter mast is used for the measurement of parameters that affect a human subject
- The 3-meter mast is used for the measurement of parameters that affect crops
- The 10-meter mast is used for the measurement of parameters without interference from objects such as trees, buildings or other obstructions
- The 30-meter mast is used for the measurement of parameters over stratified distances for the purposes of data modeling

5.3.1. Design and Implementation

Following are the important classes in the design of WSDC. Figure 5-8 represents its general architecture:

WStationDataCollection. This class generates multiple readings of *WeatherDataVector* at regular intervals, in a separate process and stores them in a queue.

WS-Transmitter. It receives *WeatherDataRequest(s)* from the *Receiver* (s), collects the observations of type *WeatherDataVector* from *WStationDataCollection* and transfers to one or multiple *WS-Receivers*.

WS-Receiver. It sends *WeatherDataRequest* to the *Transmitter* and receives *WeatherDataVector(s)*. It then decompresses the message by identifying the right compression technique. Once the *Receiver* receives the required number of observations, it can again request the *Transmitter* to transfer more weather observations at random intervals.

WSDC uses the following protocol messages (Figure 5-7):

WeatherDataVector. This data structure is passed to *WS-Transmitter* and *WS-Receiver* for exchanging weather information.

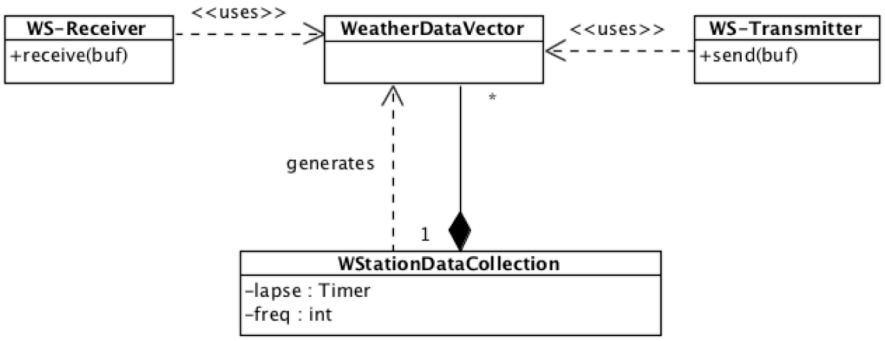

Figure 5-8: Communication of Messages between *AWS-Receiver* and *AWS-Transmitter*

WeatherDataReading. WeatherDataVector aggregates *WeatherDataReading(s).* An Instance of *WeatherDataReading* contains data, collected from different devices at a weather station.

WeatherDataRequest. WS-Receiver(s) sends *WeatherDataRequest* message to *WS-Transmitter* for receiving weather data observations. On receiving the request, *WS-Transmitter* sends all *WeatherDataVector* observations (Figure 5-9), available in *WStationDataCollection*. The *Transmitter* than goes to sleep, unless it again receives a request from the *Receiver.*

The compression control aspect creates a Quality of Service (*QoS)* monitoring channel, which runs parallel to the *WStationDataCollection*. At regular intervals, this *QoS channel* exchanges *ControlVector* that contains information about packets received and their delays. Based on the results of these control statistics, *QoSMonitor* adjusts the

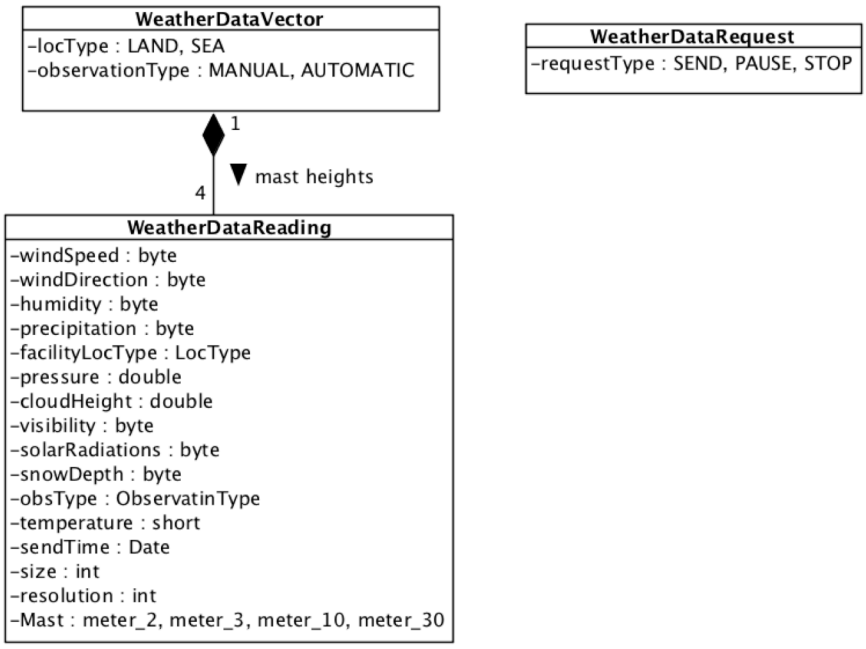

Figure 5-9: Protocol Messages for Weather Station Simulator

level of compression on the *Transmitter* and *Receiver* sides. The aspect controls the level of compression by observing the number of received messages and maximum delay per message at regular intervals using *ControlVector*. The implementation of this crosscutting concern uses the following classes (See Figure 5-10).

ControlVector. It contains compression related quality attributes that would be exchanged between *TransmitterQoS* and *ReceiverQoS*.

ReceiverQoS. At regular intervals, the *ReceiverQoS* asks *TransmitterQoS* to send *ControlVector* message, which contains control statistics about received messages and their delays.

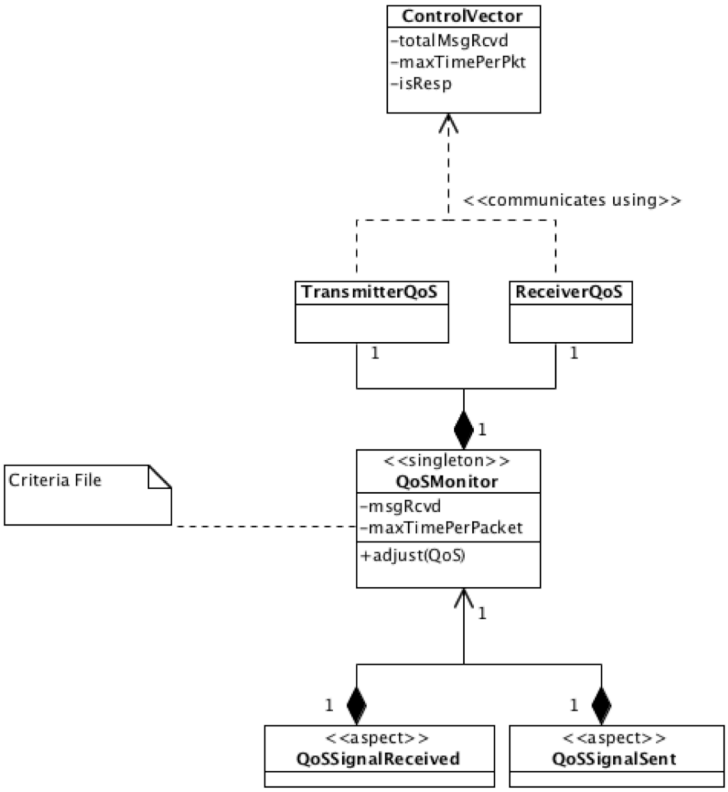

Figure 5-10: Architecture for QoS Extension

TransmitterQoS. On receiving the *ReceiverQoS* request of type *ControlVector*, it builds the control stats, updates the *ControlVector,* and retransmits the vector to *ReceiverQoS*. After sending the message, it also adjusts the *QoS* compression.

QoSMonitor. QoSCommunication channel dynamically weaves in two instances of *QoSMonitor* on the *Transmitter* and *Receiver* sides of the application. After exchanging *ControlVector* messages, *QoSMonitor(s) of QoSReceiver* and *QoSTransmitter* adjust matching compression levels for exchanging weather station observations.

In this example, *QoSSignalSent* and *QoSSignalReceived* are the two *CommJ* aspects for controlling the compression. Their code snippets are provided in Figures 5-11 and 5-12, respectively.

QoSSignalSent. It extends from reusable *OneWaySendAspect* in *CommJ*. Before sending *WeatherDataVector,* it weaves in the advice, which compresses the message with *QoSSignalReceived* (Figure 5-11).

QoSSignalReceived. It extends from reusable *OneWayReceiveAspect* in *CommJ*. After receiving the *WeatherDataVector,* it weaves in the advice, which decompresses the message with appropriate compression level matching compression level (Figure 5-12).

```
public aspect QoSSignalSent extends OneWaySendAspect{

 void around(SendEventJP _sendJp) : ConversationBegin(_sendJp){
   QoSMonitor.getInstance().statQoSReceiver();
   IMessage message = Encoder.decode(_sendJp.getBytes());

   if(message.getClass().getSimpleName().equals(WeatherDataVector.class.getSimpleName()){
        message = QoSMonitor.getInstance().getCompressMgr().compress(message);
        _sendJp.setBytes(Encoder.encode(message));
        proceed(_sendJp);
    }
  }
}
```

Figure 5-11: First Code Snippet of TurnAroundTimeAspect

```
public aspect QoSSignalRcvd extends OneWayReceiveAspect{
 private QoSMonitor qosMonitor;

 void around(ReceiveEventJP _receiveJp) : ConversationEnd(_receiveJp){
    IMessage message = Encoder.decode(_receiveJp.getBytes());
       if(message.getClass().getSimpleName().equals(WeatherDataVector.class.getSimpleName(
))){
           QoSMonitor.getInstance().statQoSTransmitter();
           qosMonitor = QoSMonitor.getInstance();
           IMessage data = (IMessage)Encoder.decode(_receiveJp.getBytes());
           qosMonitor.setTotalMsgRcvd(qosMonitor.getTotalMsgRcvd() + 1);

           for(WeatherDataReading _reading : ((WeatherDataVector)data).getReadings()){
                qosMonitor.monitorDelayInMsgs(_reading.getSendTime());
                data = qosMonitor.getCompressMgr().decompress(data);
                _receiveJp.setBytes(Encoder.encode(data));
                proceed(_receiveJp);
            }
       }
}
```

Figure 5-12: Second Code Snippet of TurnAroundTimeAspect

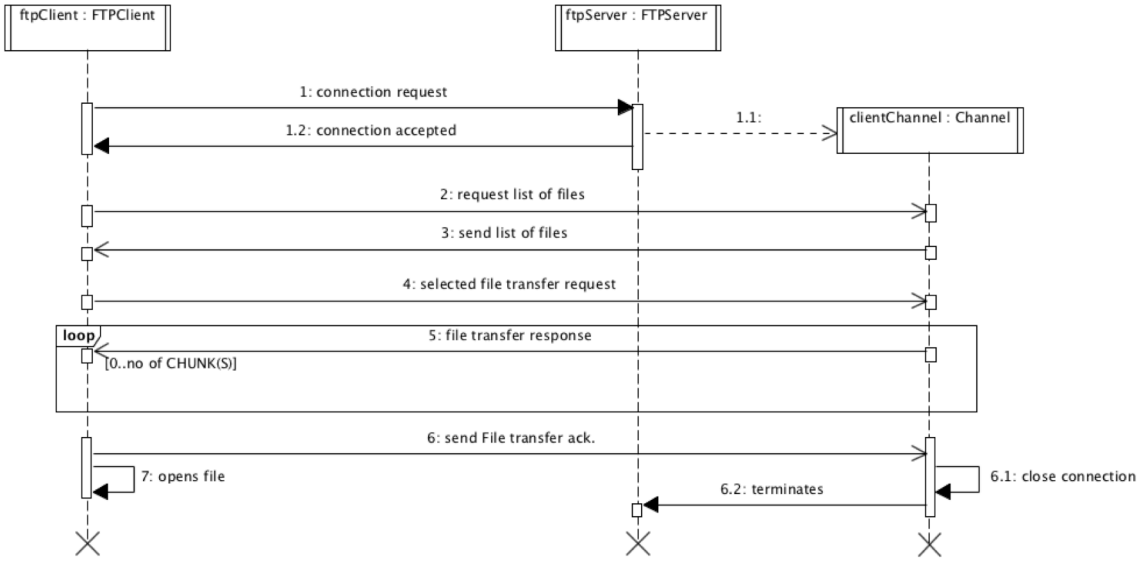

Figure 5-13: Sequence Diagram for FTP

5.4. Logging Listener and Initiator Connection Times for FTP

This section describes aspects for logging listener and initiator connection times for the processes using FTP for file transfer. Assume that an *FTPClient* establishes a *TCP* connection to an *FTPServer*. Then it requests the server for transferring a file. The

server receives the request. If the file is too big to transfer in one send, it divides the file into smaller chunks of fixed block sizes and sends each chunk with its completion status. After sending the final chunk, both the server and client close the connections.

5.4.1. Design and Implementation

As mentioned above, with *FTP*, there are two processes: an *FTPCient* and *FTPServer.* The server and client communicate using two messages, i.e., *FileTransferRequest* and *FileTransferResponse*. *FTPClient* sends a *FileTransferRequest* message to *FTPServer,* after aconnection has been established between the two processes. The *FileTransferRequest* message contains the requested file name. When *FTPServer* receives the request, it starts sending the response message (*FileTransferResponse*) to the client, which includes the file information, data chunk number and its completion status (See Figure 5-13 for more details).

Aspect - Logging Initiator Connection Time. This is an application-level connection aspect, developed using the RAL connection aspect, i.e., *CompleteConnectionAspect* (Section 4.4). It logs the time between initiating connection request to the listener *(FTPServer)* and ending of connection on the initiator (*FTPClient)* using *ConversationBeginOnInitiator* and *ConversationEndOnInitiator* pointcuts (See Figure 5-14).

Aspect - Logging Listener Connection Time. This is an application-level connection aspect, developed using RAL connection aspect, i.e., *CompleteConnectionAspect* (Section 4.4). It logs the time period between acceptance of connection request from initiator *(FTPClient)* and ending of connection on the listener

(*FTPServer)* using *ConversationBeginOnListener* and *ConversationEndOnListener*

```
public aspect InitiatorTimeAspect extends CompleteConnectionAspect{

 private long startTime = 0;
 static String timingInfo = "";

 before(ChannelJoinPoint _channelJp): ConversationBeginOnInitiator(_channelJp){
        startTime = System.currentTimeMillis();
 }

 after(ChannelJoinPoint _channelJp): ConversationEndOnInitiator(_channelJp){
        String Time  = String.format("%.3g%n",new Double(System.currentTimeMillis()
             - startTime)/1000);
        timingInfo = " Total Time of initator "
             +thisJoinPointStaticPart.getSignature().getName()+" localEP "
             + _channelJp.getConnectJp().getLocalEP()
             + " turn-around time (nano seconds) : " + Time +"\n";

  }
```

Figure 5-14: Third Code Snippet of TurnAroundTimeAspect

pointcuts (See Figure 5-15).

```
public aspect ListenerTimeAspect extends CompleteConnectionAspect{

 private long startTime = 0;
 static String timingInfo = "";

 Object around(ChannelJoinPoint _channelJp): ConversationBeginOnListener(_channelJp){

     startTime = System.currentTimeMillis();
     return proceed(_channelJp);
 }

 Object around(ChannelJoinPoint _channelJp): ConversationEndOnListener(_channelJp){

     String Time  = String.format("%.3g%n",new Double(System.currentTimeMillis()
            - startTime)/1000);
     timingInfo = " Total Time of listener "
            +thisJoinPointStaticPart.getSignature().getName()
            +" localEP turn-around time (nano seconds) : " + Time +"\n";
     return proceed(_channelJp);
 }

}
```

Figure 5-15: Fourth Code Snippet of TurnAroundTimeAspect

CHAPTER 6

MEASURING REUSABILITY AND MAINTENANCE

To measure the maintainability and reuse, we used *the Comparison Quality Model* [10] and extend it with new factors and internal attributes, forming the *Extended Quality Model* (EQM). See Figure 6-1. Section 10.4 discusses related works on measurement metrics. The EQM consisted of four parts: qualities, factors, internal attributes, and quantity metrics respectively.

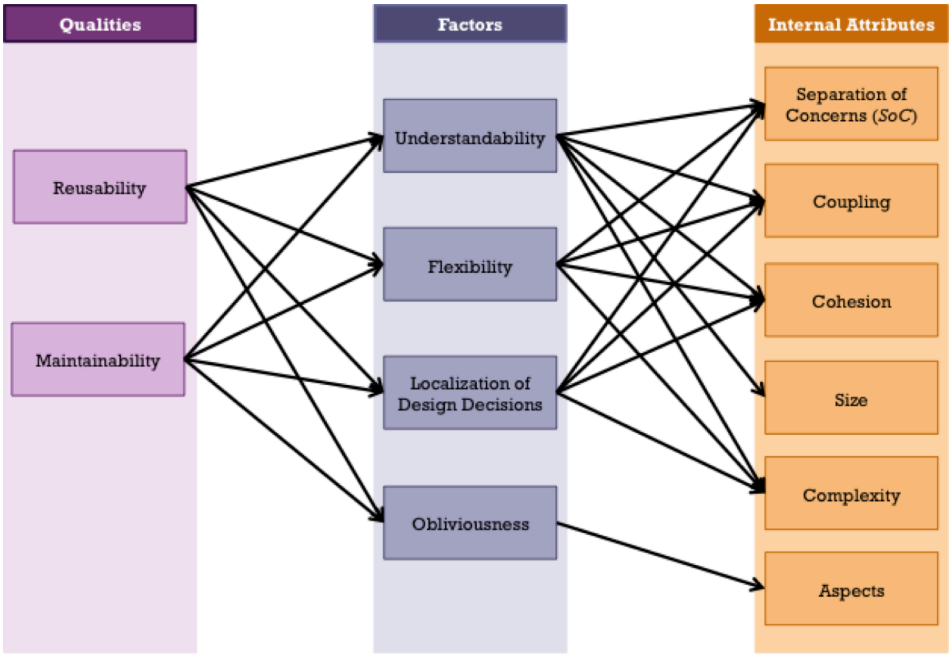

Figure 6-1: Extended Quality Model (*EQM)*

6.1. Qualities

Qualities are the attributes that we want to primarily observe in our software. They are the highest level of abstractions in our EQM and include the following:

- *Reusability*: Reusability exists for a given software element, when developers can use it for the construction of other elements or systems [24].

- *Maintainability:* Maintainability is the activity of modifying a software system after initial delivery [25]. It is the ease with which software components can be modified.

6.2. Factors

Factors are the secondary quality attributes (more granular than qualities) that influence the defined primary attributes, i.e., qualities. Following are the list of factors in our EQM.

- *Understandability:* It indicates the level of difficulty for studying and understanding a system design and code.
- *Flexibility:* It indicates the level of difficulty for making drastic changes to one component in a system without any need to change others.
- *Localization of Design Decisions:* It indicates the level of information hiding for a component's internal design decisions. Hence, it is possible to make material changes to the implementation of a component without violating the interface [26].
- *Obliviousness:* It is a special form of low coupling wherein base application functionality has no dependencies on crosscutting concerns [27].

Localization of design decisions, and code obliviousness were not part of the original quality model [7]. However, we introduced them into our EQM for two reasons. First, in his landmark paper [27], Parnas proposes three important characteristics of modular code: understandability, flexibility and localization of design decisions (information hiding). Hence, reasoning maintainability and reusability only in terms of

understandability and flexibility is not complete. Introduction of localization of design decisions is also equally important. Second, by the time Parnas proposed the definition of modular code, obliviousness had not been invented as a fundamental design principle. However, in the context of our research experiment, which depends heavily on measuring crosscutting concerns, code obliviousness becomes critical.

6.3. Internal Attributes

Internal attributes are properties of software systems related to well-established software-engineering principles, which in turn are essential to the achievement of the qualities and their respective internal factors. Following are the internal attributes in our EQM.

- *Separation of Concerns* (SoC): It defines ability to identify, encapsulate and manipulate those parts of software that are relevant to a particular concern.
- *Coupling:* It is an indication of the strength of interconnections between the components in a system. In other words, it measures number of collaborations between components or number of messages passed between components.
- *Cohesion:* The cohesion of a component is a measure of the closeness of relationship between its internal components.
- *Size:* It physically measures the length of a software system's design and code.
- *Complexity:* It measures how components are structurally interrelated to one another.
- *Tangling:* It exists when a single component includes functionality for two or more concerns, and those concerns could be reasonably separated into their own components.

- *Scattering:* It exists when two or more components include similar logic to accomplish the same or similar activities. The most serious causes of scattering occur when design decisions have not been properly localized.

6.4. Measurement Metrics

Figure 6-2 presents the metrics the EQM uses to measure each of the internal attributes. Detail descriptions of these metrics follow below.

6.4.1. SoC/Scattering Metrics

EQM includes the following metrics for SoC and code scattering: *Concern Diffusion of Application* (CDA) and *Concern Diffusion over Operations* (CDO). CDA counts the number of primary components (a class or aspect) whose main purpose is to contribute to the implementation of a concern. It counts the number of components that access the primary components by using them in attribute declarations, formal parameters, return types or method calls. CDO counts the number of primary operations

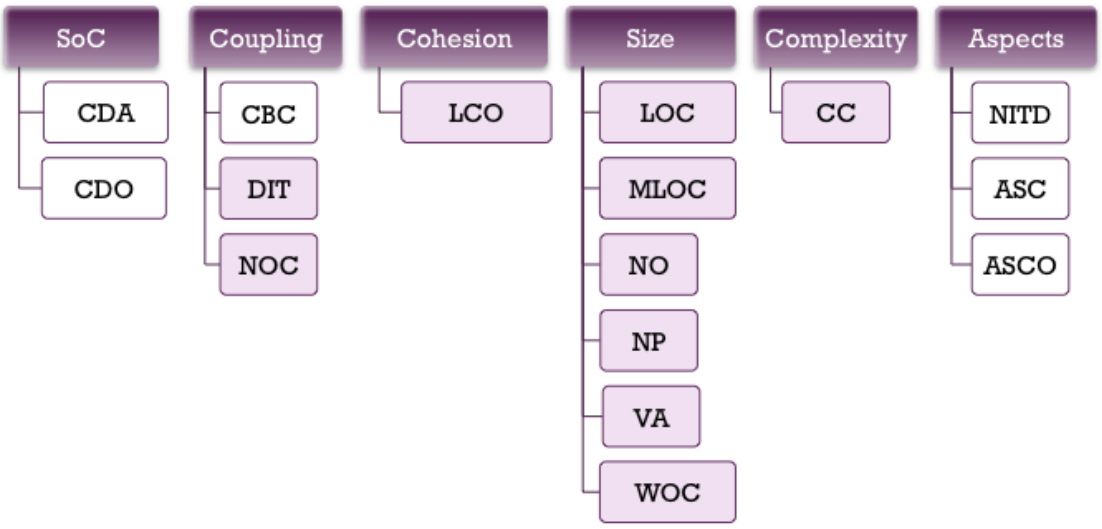

Figure 6-2: Measurement Metrics in EQM

whose main purpose is to contribute to the implementation of a concern. It also counts the number of methods and advices that access any primary component by calling their methods or using them in formal parameters, return types, and it throws declarations and local variables. Constructors also are counted as operations.

6.4.2. Coupling Metrics

Our quality model defines the following metrics for measuring coupling: Coupling between Components (CBC), Depth Inheritance Tree (DIT) and Number of Children (NOC). CBC counts the number of other classes and aspects to which a class or an aspect is coupled. On the other hand, excessive coupling of *AspectJ* concerns increases to CBC, which can be detrimental to the modular design and prevent reuse and maintenance. DIT counts how far down in the inheritance hierarchy a class or aspect is declared. As DIT grows, the lower-level components inherit or override many methods. This leads to difficulties in understanding the code and design complexity when attempting to predict the behavior of a component. NOC counts the number of children for each class or aspect. The subcomponents that are immediately subordinate to a component in the component hierarchy are termed as its children. However, as NOC increases, the abstraction represented by the parent component can be diluted if some of the children are not appropriate members of the parent component.

6.4.3. Cohesion/Tangling Metrics

Our quality model defines the following metrics for measuring cohesion and tangling among components: *Lack of Cohesion in Operations* (LCO).

LCO measures the lack of cohesion of a class or aspect in terms of the amount of method and advice pairs that do not access the same instance variable. If the related methods do not access the same instance variable, they logically represent unrelated components and hence should be separated.

6.4.4. Size Metric

Our quality model defines the following size metrics: *Lines of Code (LOC), Method Lines of Code* (MLOC)*, Number of Operations* (NO)*, Number of Parameters* (NP)*, Vocabulary Size* (VA) and *Weighted Operations per Component* (WOC).

LOC counts the lines of code. The greater the LOC, the more difficult it is to understand the system and harder to manage the software maintenance activities or understand the implementation of the required functionalities during maintenance and reuse activities. MLOC counts the method lines of code. Kremer [23] states that the greater the average of MLOC for a component, the more complex the component would be. NO counts the number of operations in a component. Objects with large number of operations are less likely to be reused. Some times LOC is less but NO is more, which indicates that the component is more complex. NP counts the number of parameters for methods in each class or aspect. NP is an Operation-Oriented Metric. A method with more parameters is assumed to have more complex collaborations and may call many other method(s). VA counts the number of system components, i.e., the number of classes and aspects into the system. Sant' Anna [7] points out that if number of components increase, it is an indication of more cohesive and less tangled set of ADT.

Finally, WOC metric measures the complexity of a component in terms of its operations. WOC does not specify the operation complexity measure, which should be

tailored to the specific contexts. The operation complexity measure is obtained by counting the number of parameters of the operation, assuming that an operation with more parameters than another is likely to be more complex. It is an object-oriented design metric, proposed by Kemerer [23] and sums up the complexity of each method. The number of methods and complexity is an indication of how much time and effort is required to develop and maintain the object. The larger the value of weighted operations, the more complex the program would be.

6.4.5. Complexity Metric

Our quality model defines the following complexity metrics: *McCabe's Cyclomatic Complexity* (CC) [28]. Mathematically, the cyclomatic complexity of a structured program is defined with reference to the control flow graph of the program, a directed graph containing the basic blocks of the program, with an edge between two basic blocks if control may pass from the first to the second. The complexity *M* is then defined as:

$$M = E - N + 2P$$

Where:

E = the number of edges of the graph

N = the number of nodes of the graph

P = the number of connected components (exit nodes).

CC measures the logical complexity of the program. The metric defines the number of independent paths and provides you with an upper bound for the number of test cases that must be conducted to ensure that all statements have been executed at least once. High value of CC affects program maintenance and reuse.

6.4.6. Obliviousness Metric

Our quality model defines the following obliviousness metrics: *Number of Inter-type Declarations* (NITD)*, Aspect Scattering Over Components* (ASC)*, Aspect Scattering Over Component Operations* (ASCO)*.* NITD counts the number of inter-type declarations. A higher value of NITD indicates a tighter coupling between the aspect and application components. ASC counts the number of aspect components scattered over application components. It measures the tangling of aspects in the application components. More tangling of aspects in the program makes the original application less reusable and maintainable. ASCO counts the number of aspect components scattered over application component operations. ASC (discussed above) gives a high-level overview of the application tangling in the aspect components but ASCO provides more insight on operations-level tangling of applications inside aspect components.

CHAPTER 7

HYPOTHESES

To determine whether *CommJ* improves reusability and maintainability, I conducted an experiment that tests the seven hypotheses listed below. All of these hypotheses have the same premise and refer to the metrics defined for the EQM described in Chapter 6.

Hypothesis #1: If crosscutting communication concerns are effectively encapsulated in *CommJ* aspects, then the software has better separation of concerns and less scattering (as described by CDA, CDO in Section 6.4.1.) than equivalent systems developed with AOP design techniques.

Method of Calculation:

- CDA. Counts the total lines of concern-related occurrences in an application level component. Concern occurrences can be in an aspect or a class. It is a manual calculation.
- CDO. Counts the total number of operations in an application-level component containing the concern related occurrences. It is a manual calculation.

Prediction: For this hypothesis to hold, we expect that CDA, CDO will decrease when using *CommJ.*

Hypothesis #2: If crosscutting communication concerns are encapsulated in *CommJ* aspects, the software has lower coupling (as described by CBC, DIT, NOC in Section 6.4.2.) than equivalent systems developed with AOP design techniques.

Method of Calculation:

- NOC. Describes the total number of direct subcomponents of a component. Additionally, if a component is implementing an interface, it counts as a direct child of that interface. The tool [29] calculates this metric.
- CBC. Counts the total number of associations, dependencies between components of a program. It is a manual calculation.
- DIT. Maximum hierarchical distance from component object in the inheritance hierarchy. It is a manual calculation.

Prediction: For this hypothesis to hold, we expect that NOC, CBC, DIT will decrease when using *CommJ.*

Hypothesis #3: If crosscutting communication concerns are encapsulated in *CommJ* aspects, the software has higher cohesion and less tangling (as described by LCO in Section 6.4.3.) than equivalent systems developed with AOP design techniques.

Method of Calculation: LCO. Measures for the cohesiveness of a component and is calculated with the Henderson-Sellers method. If (m(A) is the number of methods accessing an attribute A, it calculates the average of m(A) for all attributes, subtracts the number of methods m and divides the result by (1-m). A low value indicates a cohesive component, and a value close to 1 indicates a lack of cohesion and suggests the component might better be split into a number of (sub) components. The tool [29] calculates this metric.

Prediction: For this hypothesis to hold, we expect that LCO will decrease for *CommJ.*

Hypothesis #4: If crosscutting communication concerns are encapsulated in *CommJ* aspects, the software is not significantly larger (as described by LOC, MLOC, NO, NP, VA, WOC in Section 6.4.4.) than that of equivalent systems developed with AOP design techniques.

Method of Calculation:

- LOC: It counts the total lines of code excluding white spaces and comments. The tool [29] calculates this metric.
- MLOC: It counts the total lines of code for a method or advice ignoring white spaces and comments. The tool [29] calculates this metric.
- NO: It counts the total number of operations in a component. The tool [29] calculates this metric.
- NP: It counts the total number of parameters for all methods in a component. The tool [29] calculates this metric.
- VA: It counts the total number of components, which include classes, aspects, and inner classes. The tool [29] calculates this metric.
- WOC: It sums up the CC for all methods in a component. The tool [68] calculates this metric.

Prediction: For this hypothesis to hold, we expect that:

- LOC, MLOC, NO, NP, VA, WOC will decrease, and
- VA will increase for *CommJ.*

Hypothesis #5: If crosscutting communication concerns are encapsulated in *CommJ* aspects, the software is not significantly complex (as described by CC in Section 6.4.5.) than equivalent systems developed with AOP design techniques.

Method of Calculation: CC: Counts the number of flows through a piece of code. Each time a branch occurs (if, for, while, do, case, catch and the ?: ternary operator, as well as the && and || conditional logic operators in expressions) this metric is incremented by one. It is calculated for methods/advice only. The tool [29] calculates this metric.

Predictions: For this hypothesis to hold, we expect that CC will decrease when using *CommJ.*

Hypothesis #6: If crosscutting communication concerns are encapsulated in *CommJ* aspects, the software is significantly more oblivious (as described by NITD, ASC, ASCO in Section 6.4.6.) than equivalent systems developed with AOP design techniques.

Method of Calculation:

- NITD: It counts the number of inter-type declarations in the aspects and number of times they are used, which also includes their references in the aspects and application classes. It is a manual calculation.
- ASC: It counts the number of distinct application components in the concerns, which includes both the distinct number of components and number of operations for those components. It is a manual calculation.
- ASCO: It counts the number of methods and advices in the concern containing the references of application components. It is a manual calculation.

Prediction: For this hypothesis to hold, we expect that NITD, ASC, ASCO will decrease when using *CommJ.*

Hypothesis #7: If crosscutting communication concerns are encapsulated in *CommJ* aspects, the extension part of the software requires less number of changes to reuse and maintain (as measured by Eclipse IDE diff function) than equivalent systems developed with AOP design techniques.

Method of Calculation:

- CR: Number of changes required to reuse the concern for another application. The eclipse IDE calculates this metric.
- CM: Number of changes required to maintain the concern. The eclipse IDE calculates this metric.

Prediction: For this hypothesis to hold, we expect that the number of changes to reuse and maintain will decrease when using *CommJ.*

CHAPTER 8

EXPERIMENTAL METHOD

The experiment to test the previously stated hypotheses consists of the 17 general steps listed below. Additional details about the more complex steps can be found in Sections 8.1 through 8.7. Section 8.8 discusses the independent and dependent variables. Further, Section 8.9 describes how I minimized threats to validity caused by extraneous variables.

Preliminaries

1. All the researchers passed the online Human Research Training course offered through the Collaborative Institutional Training Initiative (CITI). See Appendix I for more details.
2. Submitted an application for a Human Research Experiment to the Institutional Review Board (IRB) and got its approval (See Appendix I for more details).
3. Developed three simple software applications and documented their requirements, design, and implementation. See Section 8.1 for more details.
4. Selected three common communication-related crosscutting concerns for the above sample applications. Developed an initial requirements specification document. See Section 8.2 for more details.
5. Sent invitation letters (See Appendix I) and recruited seven volunteer developers who were experienced in object-oriented software development (Section 8.3.1), and randomly organized them into two study groups: A and B. Group A programmed using a AOP approach and Group B used *CommJ*.

6. Had the seven volunteers complete a survey that assessed their background and skill levels (Appendix C). See Section 8.3.3.
7. Provided AOP training to developers in Group A, and had them worked through some practice applications. See Section 8.4.
8. Provided *CommJ* training to developers in Group B, and had them worked through some practice applications. See Section 8.4.

Phase 1

9. Gave three sample applications mentioned above, associated documentation (Appendix A), and all three concerns initial requirements specifications (Appendix B) to the seven developers.
10. Asked them to complete a pre-implementation questionnaire (Appendix D), once they understood the code and documentation provided to them in Steps 7, 8 and 9.
11. Asked them to develop the three crosscutting concerns, and collected their implementations. See Section 8.5.
12. Asked volunteers to complete a post-questionnaire that gathered additional information to measure quality metrics. See Appendix D.
13. Measured the quality metrics using EQM, collected findings from the logs and post/post-questionnaires from Phase 1.

Phase 2

14. Gave enhancements (sample applications and crosscutting concerns) to all seven developers, had them revise their implemented concerns, and then collected those revised implementations. See Section 8.6.
15. Asked them to complete a questionnaire (Appendix G) that gathered additional information to measure quality metrics.
16. Evaluated the reusability and maintainability of the various software artifacts using EQM. See Section 8.7 for details on the metrics and experiment.
17. Interrupted the results.

Section 8.8 summarizes the control, independent, dependent, and extraneous variables for this experiment. Section 8.9 describes possible threats to validity of the research experiment.

8.1. Selection of Sample Applications

Table 2 describes three selected applications for the experiment. To improve the validity of the experiment, it was important that the sample applications were non-trivial systems and that their communications represented a broad range of issues. To this end, the sample applications were all multithreaded, used JDK sockets or channels, included different types of communication heterogeneities (Section 2.2.), had one or more senders, and contained opportunities for different types of conversations. Developers were provided with the application code along with their documentation and UML diagrams.

Table 2. Selected sample applications

Application Name	Description
Levenshtein Edit-Distance Calculator (LD)	The programmer implemented an application where a server would calculate the LD between two input strings, provided by the client, over a connection-oriented communications.
File Transfer Program (FTP)	The programmers implemented a file transfer protocol over connection-oriented communication.
Weather Station Simulator (WS)	The programmers implemented a simple weather station simulator, supported by a Transmitter and a Receiver.

Table 3. Selected sample crosscutting concerns

Aspect Name	Description
Version Compatibility	This concern adapted one version of the message to another, so processes running different versions could still communicate with each other. The crosscutting concern included knowledge of converting one version to another and conversely
Symmetric-Key Encryption	It encrypted the communication between a sender and receiver using symmetric-key encryption
Measuring Performance	It measured some performance related statistics for message-based communications between a sender and receiver

8.2. Selection of Crosscutting Concerns from Sample Applications

We selected the crosscutting concerns for the experiment such that they could apply to all the sample applications and the various types of conversations described in Section 4.4. Additionally, these concerns needed to be sufficiently simple that a novice programmer (i.e., one who meets the criteria specified in Section 8.3) could integrate them into the sample applications in less than 10 hours, regardless of whether *CommJ* is used. Table 3 describes the three crosscutting concerns selected for the experiment. Appendix B provides more details about these selected crosscutting concerns.

8.3. Recruitment of Developers

8.3.1. Criteria for Selection of Developer

All participants were either undergraduate or graduate students in computer science. They had taken courses in algorithms, data-structures, *Java* and software engineering. They had also good exposure of OOD and Unified Modeling Language (UML). In addition, they had implemented at least one multi-threaded network programming project using *Java*, and the size of the project was comparable to the scope of our implementations.

8.3.2. No Personal-Identifying Information

Once selected, each volunteer was assigned a unique number. Data and code gathered from the volunteers were tagged with this number. No other identifying information was collected. Furthermore, we kept no record of the volunteers' assigned numbers.

8.3.3. Survey to Assess Skill Levels

To identify the effects of extraneous variables (Section 8.9), developers were asked to fill a questionnaire after hiring and before starting the experiment. The results of this survey, provided in full in Appendix H, clearly indicate that our selection of candidates fulfilled all the criteria mentioned in Section 8.3.1.

8.4. Training of Developers

After organizing the participants into two groups, Group A developers were trained on how to write aspects using *AspectJ*, and the Group B developers were given training for both *AspectJ* and *CommJ*. During training, each developer implemented three

sets of examples, similar to those that would be part of the experiment. Later results from the pre-implementation questionnaires (Appendix D) reveals that 100% of the developers found these questionnaires very helpful in understand and coding the language related complications.

8.5. Developing Crosscutting Concerns Using Initial Set of Requirements and Collected Artifacts

All seven developers were given an initial set of requirements in which they were asked to implement three concerns using sample applications (Sections 8.1 and 8.2).

During this phase, we found that correctly understanding the requirements, familiarity with the language, and debugging were the three most prominent challenges. First, on requirements understanding, 42% of the total participants agreed that understanding and analyzing the requirements correctly was the most time consuming activity in Phase 1, whereas none of the participants complained about this during second phase. Second, 57% of the total participants said that familiarity with the language/tool was the hardest thing during initial phase of implementation, whereas no participant raised this issue again in the second phase. Third, debugging for both *AspectJ* and *CommJ* took more time than initial development. Specifically, 57% of the participants supported this observation in Phase 1, and 71% supported it again in Phase 2. This observation indicates that debugging time may be more connected to the complexity of the requirements than to experience with the implementation platform.

8.6. Extended Set of Requirements and Collected Artifacts

Once the developers had implemented the requirements in Section 8.5 and we calculated the code metrics, the developers were given an extended set of requirements for the crosscutting concerns, updated sample application codes, and revised descriptions.

Overall participants found that in this phase, their debugging time increased (from 57% in the initial phase to 71% in the second phase). Neither understanding the requirements nor familiarity with the language/tool presented a significant issue, and developers spent much less time to implement the requirements, compared to the initial phase. Specifically, 86% of the participants confirmed that they spent almost 50% less time to implement the Phase 2 requirements, compared to the Phase 1 requirements.

8.7. Measuring Dependent Variables Using Reuse/Maintainability Metrics

I measured EQM code metrics (Section 8.7), using both manual- and tool-based [65] methods. Total measurements include following:

- Experiment input variables included a total of seven developers, three applications with two versions each.
- Experiment generated a total of 28 software systems against which the metrics need to be applied.
- The 16 code metrics of EQM required a total of 448 measurements. Of these 448 measurements, 280 measurements from 10 metrics were generated using tools, and 168 measurements from 6 metrics were calculated manually.

After data collection using the above code metrics measurement procedure, we interpreted our hypothesis (Chapter 7) using the dependent variables in Section 8.8.

8.8. Independent and Dependent Variables

For this experiment, the only independent variable was the implementation method. It had two possible values (i.e., AOP, and *CommJ*).

The dependent variables were those that we wanted to observe possible difference among the groups. All instruments in our EQM (Chapter 6) represented our dependent variables.

- Measurement metrics (Section 6.4) were our direct independent variables
- Internal attributes (Section 6.3) were indirect dependent variables, which were interpreted from measurement metrics
- Factors (Section 6.2) were indirect dependent variables and were interpreted by using internal attributes
- Finally, qualities (Section 6.1) were indirect dependent variables and were interpreted by using factors

8.9. Extraneous Variables and Mitigation of Threats to Validity

Extraneous variables were other factors that might affect the dependent variables being studied, but were difficult or impossible to control. Below is a list of extraneous variables (threats to validity) in our research experiment, along with our mitigation strategies to control their effects on the research experiment output.

- *Development Experience.* Our selection criteria for hiring the developers (Section 8.3.1), and survey to assess their skill levels (Section 8.3.3) reasonably mitigated its effect.

- *Capacity to Work.* Training of developers (Section 8.4) for specialized skills, needed in this experiment, reasonably mitigated the effect of this extraneous variable.
- *Intelligence.* We found no sufficient mitigation strategy to control this threat.
- *Health Factors.* We found no sufficient mitigation strategy to control this threat.
- *Work Environment.* We found no sufficient mitigation strategy to control this threat.
- *Personnel Commitment of Developers for Better Design.* We found no sufficient mitigation strategy to control this threat.
- *Accuracy in Manual Measurements.* More than one people measured the metrics.
- *Accuracy in Tool's Measurements:* Human resources were asked to manually calculate measurements using EQM metrics, which crosschecked the tool's automatically-generated measurement with manual ones and hence effectively mitigated the inaccuracy risks.

CHAPTER 9

RESULTS AND INTERPRETATIONS

9.1. Separation of Concerns

Hypothesis #1 theorized that if crosscutting communication concerns are effectively encapsulated in *CommJ* aspects, the software has better separation of concerns and less scattering as measured by CDA and CDO than equivalent systems developed with AOP design techniques. In other words, the CDA and CDO metric values for *CommJ* should be less than *AspectJ* (See Section 6.4.1. for details on metrics). We found CDA and CDO did decrease for the *CommJ* group. In Figures 9-1 and 9-2, the vertical axes represent the CDA and CDO measurements, and the horizontal axes represent the four activities of the experiment. For each activity there are two bars: a blue bar for the results of *AspectJ* group and a green bar for the results of *CommJ* group.

Not only were CDA and CDO values reduced using *CommJ*, but also they went to zero in all four activities of the experiment. The reason for phenomena is that *CommJ* pointcuts provide total obliviousness between the application and communication-related crosscutting concern. In *AspectJ,* components and their operations for crosscutting

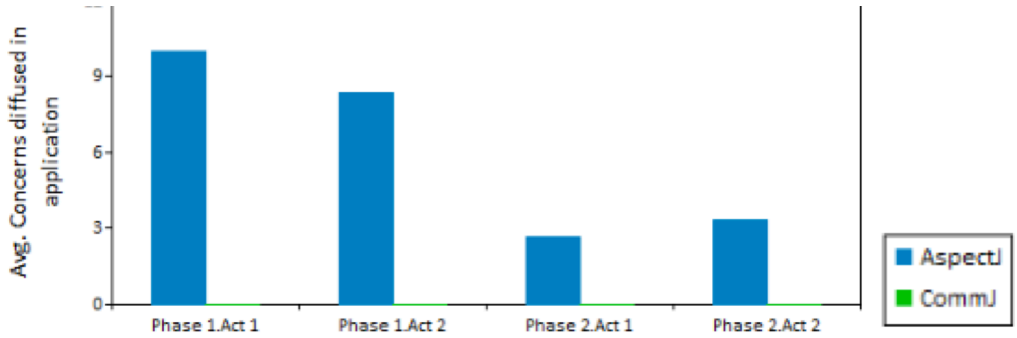

Figure 9-1: CDA Coverage over Phases

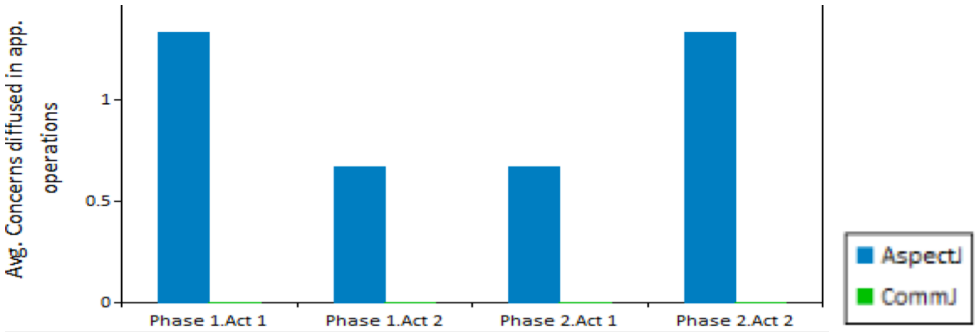

Figure 9-2: CDO Coverage over Phases

concern were significantly more diffused in the application because the pointcuts had to be tied to programming constructs instead of communication abstractions.

From these results, we can confidentially conclude that Hypothesis#1 holds true for better separation of concerns in *CommJ* implementations than in *AspectJ*.

9.2. Coupling

Hypothesis #2 theorized that if crosscutting communication concerns are effectively encapsulated in *CommJ* aspects, the software has lower coupling as measured by CBC, DIT and NOC than equivalent systems developed with AOP design techniques. In other words, CBC, DIT and NOC metric values for *CommJ* should be less than *AspectJ* (See Section 6.4.2. for details on metrics). Figures 9-3 thourgh 9-5 indicate that *CommJ* implementations significantly reduced the values of CBC, DIT and NOC, respectively, as compared to *AspectJ* implementations in all the four phases of the experiment. *CommJ* crosscutting concerns did not maintain any direct relationship with the application components and thus had a lower CBC value. However, in *AspectJ*, excessive coupling of concern with the application increased CBC, which hindered reuse and maintenance.

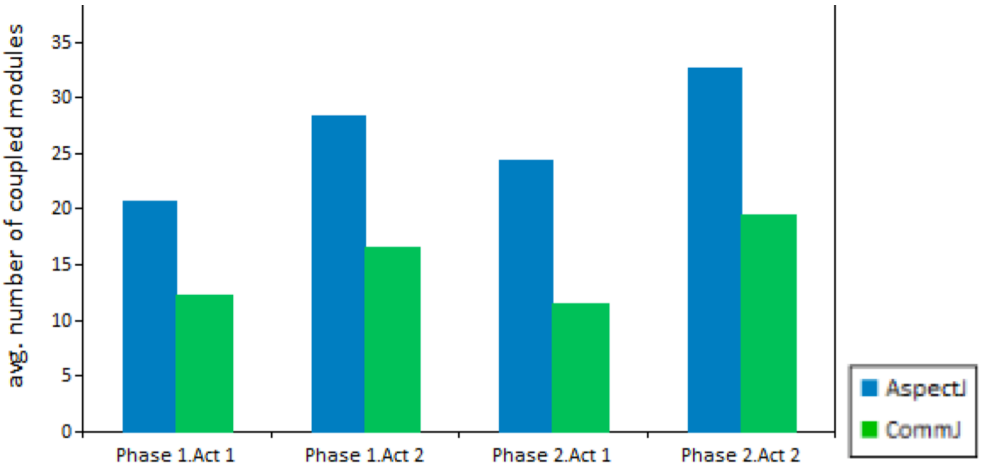

Figure 9-3: CBC Coverage over Phases

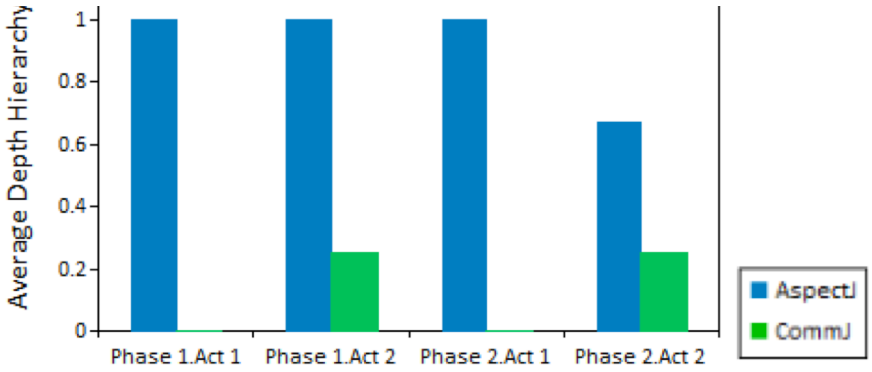

Figure 9-4: DIT Coverage over Phases

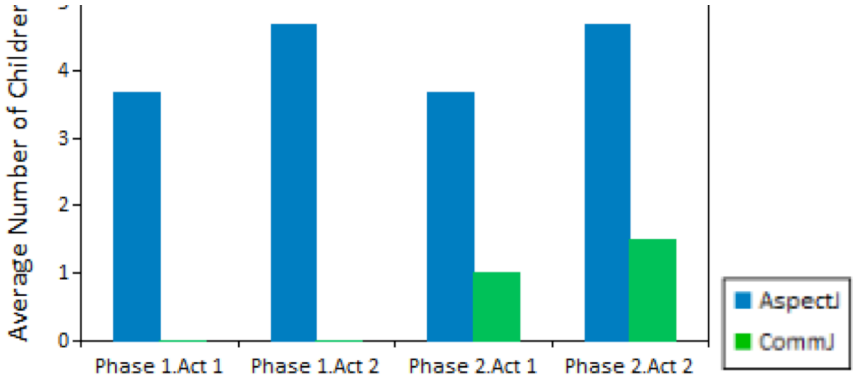

Figure 9-5: NOC Coverage over Phases

The reason for higher DIT and NOC values in *AspectJ* was that the participants preferred to override parent methods in crosscutting concerns to share data structures across aspect and application components during message passing. However, *CommJ* provides a comprehensive set of pointcuts, which fully encapsulates the IPC abstractions, and thus participants did not need to override or inherit the aspect components. From these results, we can confidentially conclude that Hypothesis#2 holds true for reduced coupling in *CommJ* than in *AspectJ*.

9.3. Cohesion

Hypothesis #3 theorized that if crosscutting concerns are effectively encapsulated in *CommJ* aspects, the software has higher cohesion (as described by LCO in Section 6.4.3.) than equivalent systems developed with AOP design techniques. In other words, the LCO metric value for *CommJ* should be less than *AspectJ*. The results shown in Figure 9-6 demonstrate that *CommJ* maintains a lower value for LCO than *AspectJ* in all four phases of the experiment. Santana [10] says that LCO measures the degree to which a component implements a single logical function. Results proved that *CommJ*

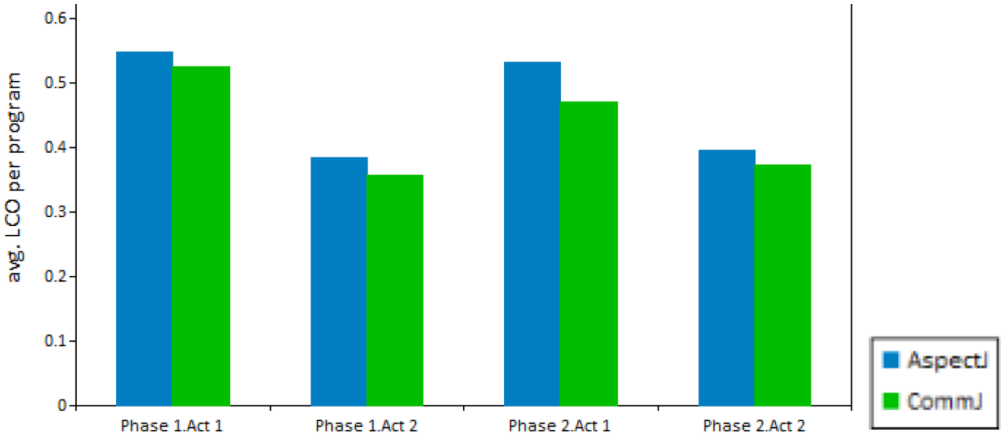

Figure 9-6: LCO Coverage over Phases

implementations were more cohesive and logical than *AspectJ*, hence have a lower LCO value.

From these results, we can confidentially conclude that Hypothesis#3 holds true for increased cohesion in *CommJ* than in *AspectJ*.

9.4. Size

Hypothesis #4 theorized that if crosscutting communication concerns are effectively encapsulated in *CommJ* aspects, the software is not significantly larger (as described by LOC, MLOC, NO, NP, WOC, VA in Section 6.4.4.) than equivalent systems developed with AOP design techniques. In other words, LOC, MLOC, NO, NP, WOC metrics values for *CommJ* should be less and VA be more than *AspectJ*. Figures 9-7 through 9-11 show that *CommJ* implementations significantly reduced the metrics values for LoC, MLoC, NP, NO and WOC in all phases of the experiment.

In comparison with *AspectJ*, *CommJ* participants found a more neat and clean set of pointcuts in IPC abstractions, which helped them to code the crosscutting concerns in less LOC. *CommJ* conceptually models various general network and distributed

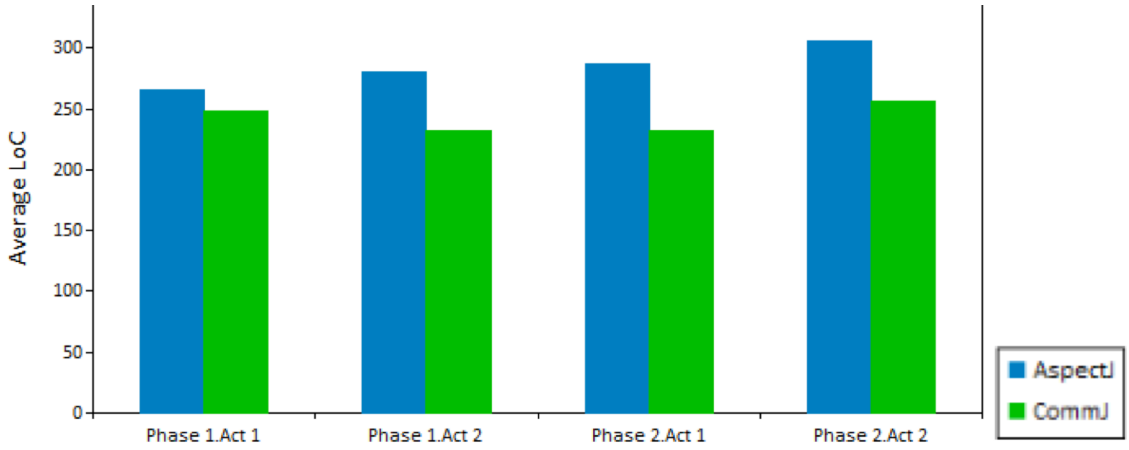

Figure 9-7: Average LoC Coverage over Phases

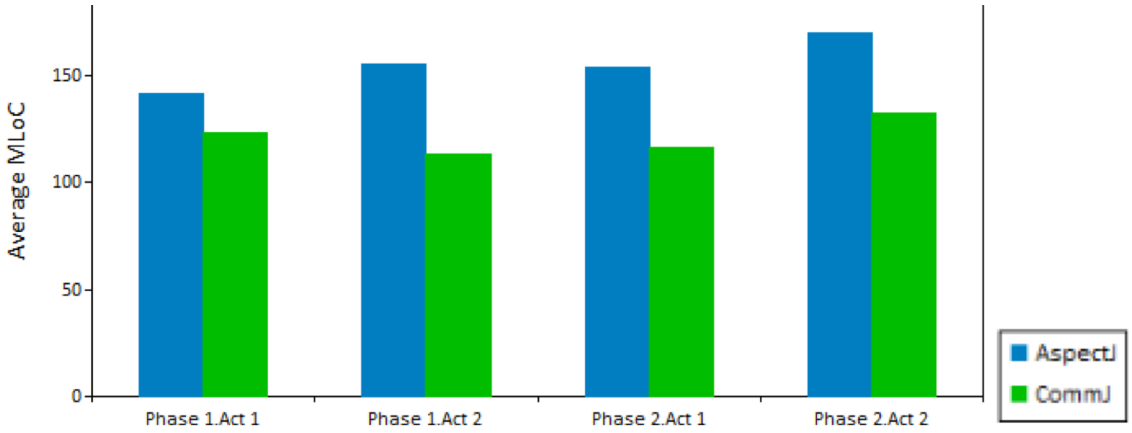

Figure 9-8: Average MLoC over Phases

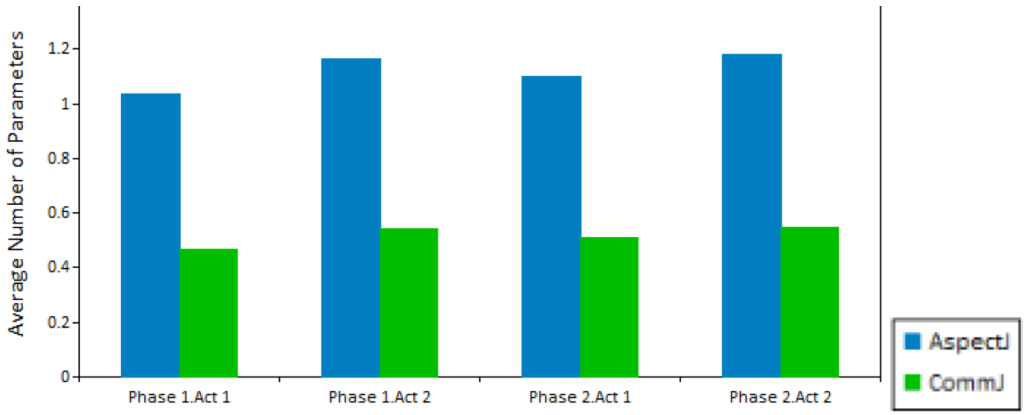

Figure 9-9: Average NP over Phases

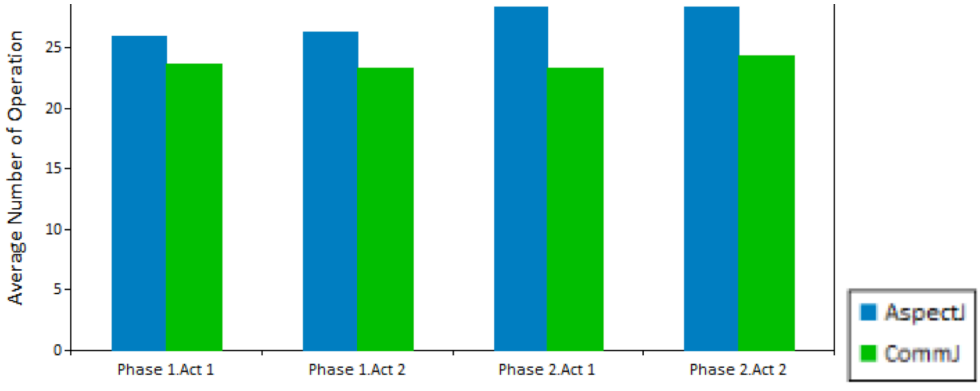

Figure 9-10: Average NO over Phases

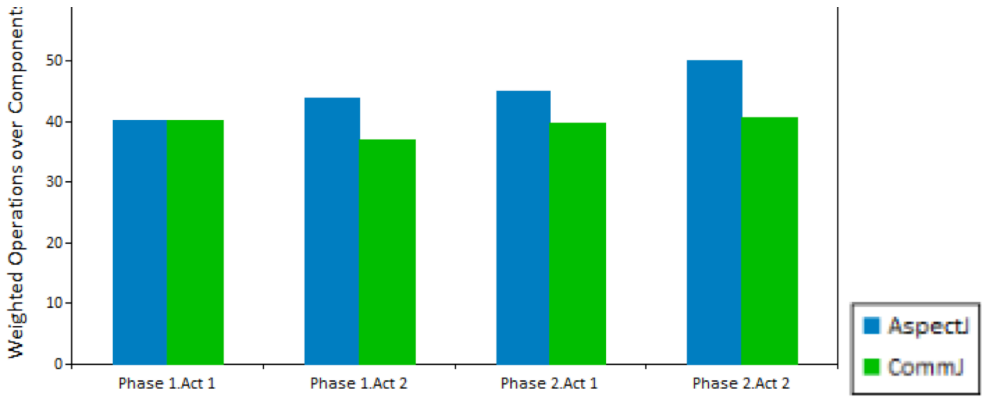

Figure 9-11: Average WOC over Phases

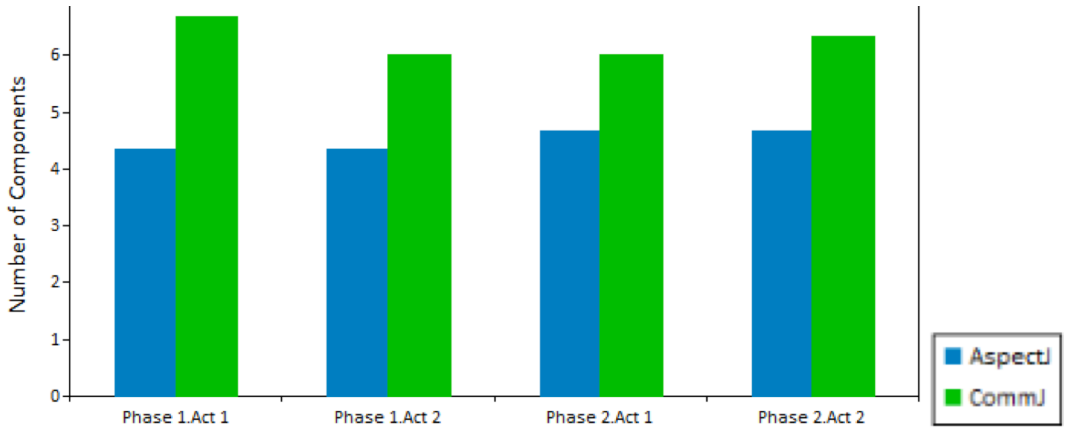

Figure 9-12: Average VA over Phases

abstractions using UMC (Section 4.1) into rich set of communication and connection join points along with general purpose family of conversations, which helped the participants to implement the application crosscutting concerns in simpler and more logical method bodies, with no extra lines of code and less number of operations. Hence it reduced MLOC, NO, NP and WOC.

As predicted by the above hypothesis, results shown in Figure 9-12 give sufficient evidence that average VA for all programs was more for *CommJ* than *AspectJ*. Although

the number of components were more in *CommJ* implementations, they were more cohesive.

From these results, we can conclude that Hypothesis#4 holds true for improved code size in *CommJ* than in *AspectJ*.

9.5. Complexity

Hypothesis #5 theorized that if crosscutting communication concerns are effectively encapsulated in *CommJ* aspects, the software is significantly less complex (as described by CC in Section 6.4.5.) than equivalent systems developed with AOP design techniques. In other words the CC value for *CommJ* should be less than *AspectJ*. Figure 9-13 shows that the value of CC is smaller for *CommJ* than *AspectJ*, because *CommJ* hides complex IPC abstractions, which result in simple conditional statements and less tangled code.

From these results, we can confidentially conclude that Hypothesis#5 holds true for less complex software in *CommJ* than *AspectJ*.

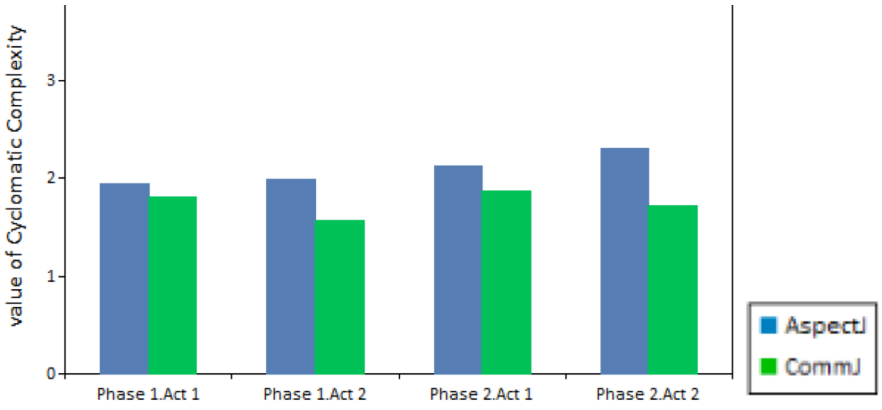

Figure 9-13: Average CC over Phases

9.6. Obliviousness

Hypothesis #6 theorized that if crosscutting communication concerns are effectively encapsulated in *CommJ* aspects, the software will be more oblivious (as described by NITD, ASC, ASCO in Section 6.4.6.) than equivalent systems developed with AOP design techniques. In other words, NITD, ASC, ASCO for *CommJ* should be less than *AspectJ*. Figures 9-14 through 9-16 show that *CommJ* implementations significantly reduced the values of NITD, ASC and ASCO metrics.

In comparison with *AspectJ*, the reason for having a zero value for NITD in *CommJ* was that the participants used IPC constructs and did not need to use inter-type declarations (ITD) for sharing of data structures between application and aspect component. Significant reduction in ASC and ASCO was due to the layers of indirection

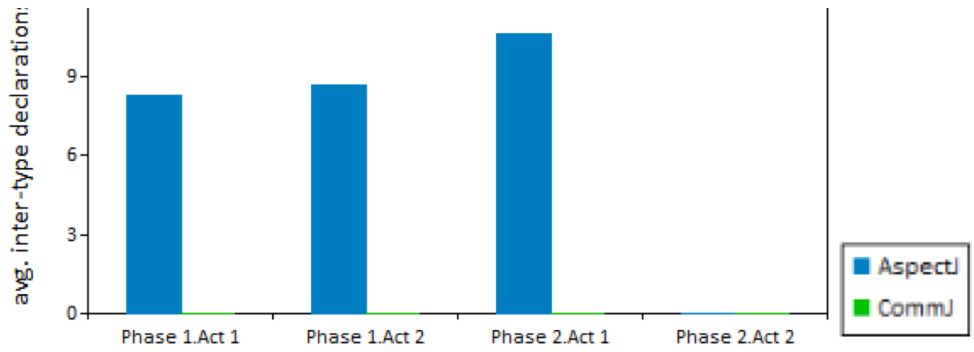

Figure 9-14: Average NITD over Phases

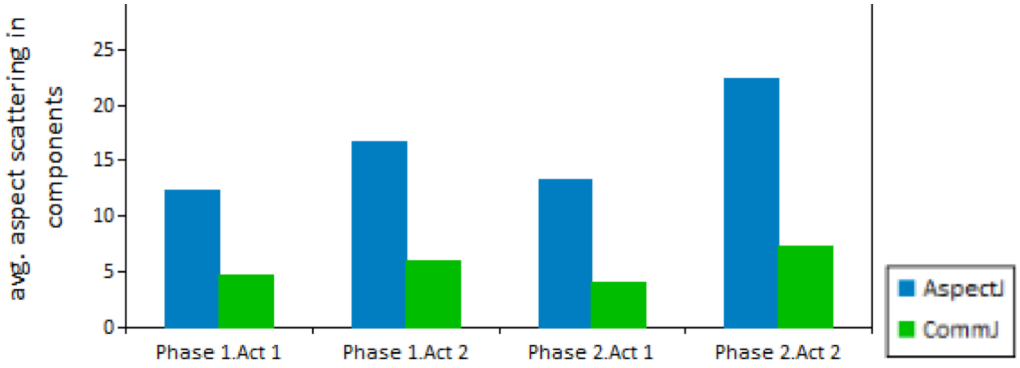

Figure 9-15: Average ASC over Phases

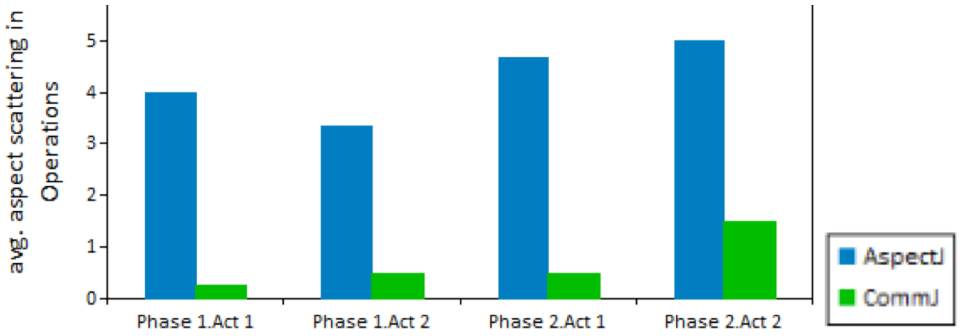

Figure 9-16: Average ASCO over Phases

between the application and aspect components, which CommJ provides but are missing in AspectJ.

From these results, we can confidentially conclude that Hypothesis#6 holds true for less oblivious software concerns in *CommJ* than *AspectJ*.

9.7. Reuse and Maintenance of Concern

Hypothesis #7 theorized that if crosscutting communication concerns are effectively encapsulated in *CommJ*, the crosscutting concern will require a smaller number of changes in order to reuse and maintain (as measured by CR, CM in Chapter 7) than equivalent systems developed with AOP design techniques. In other words CR, CM values for *CommJ* should be less than *AspectJ*. From the results shown in Figure 9-17,

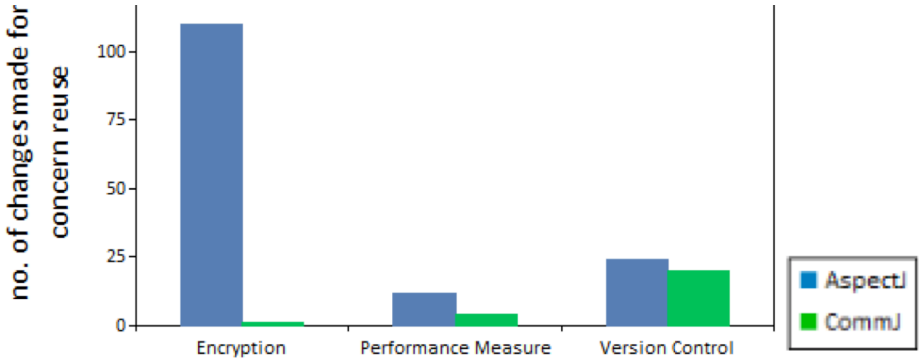

Figure 9-17: CR over Extensions

we can see that *CommJ* implementation significantly reduced the changes required to reuse the previous implementations in the second phase of the experiment than *AspectJ*.

CommJ aspects were overall more oblivious, logical and independent from the base application than *AspectJ* concerns and so they reduced the CR value in all four phases of the experiment.

Figure 9-18 provides another graphical representation to analyze reuse for *AspectJ* and *CommJ*. The light green colored-graphs represent scattering in *CommJ* (aspects only) and light blue colored-graphs represent *AspectJ* implementations. The scattered points in graph indicate the number of changes for reusing a concern with *CommJ* and *AspectJ* in different activities of Phases 1 and 2, respectively. The scattered points in blue represent

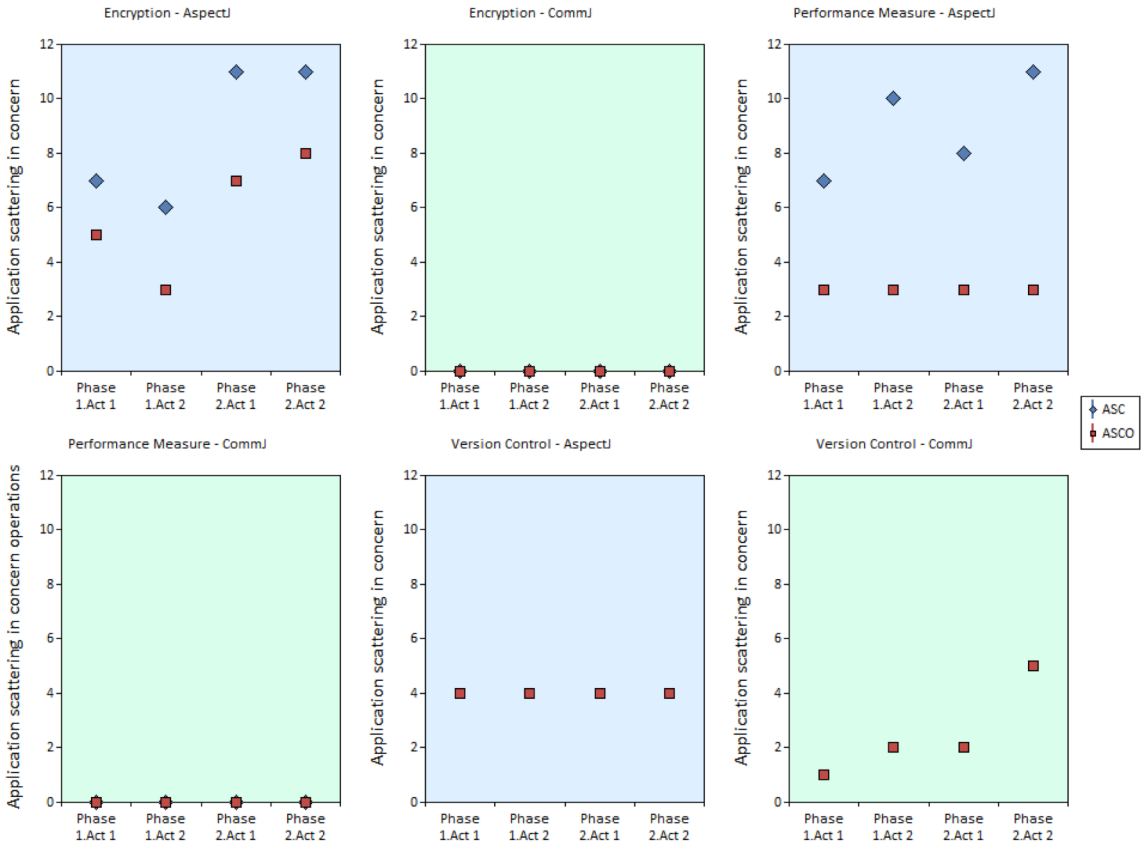

Figure 9-18: ASC and ASCO over Phases in *AspectJ* and *CommJ*

ASC and in red represent ASCO metrics results. Overall, the results of the graph indicate that ASC and ASCO remained zero for all the activities of *CommJ* (highly reusable), but it was highly scattered in *AspectJ*. The reason for less scattering is discussed in Section 9.6 above.

Figure 9-19 shows the number of changes required to maintain the program in its initial activity (Activity 1 of Phase 1) to its maintenance activity (Activity 2 of Phase 2), reduced significantly for *CommJ* than *AspectJ*. The difference between CR and CM is that in CR we are only considering changes in the concern; however, in CM we are interested in number of changes both in the concern and application. We found that *CommJ* concerns were overall more oblivious, logical and independent from the base application than *AspectJ* concerns, and so they have reduced CM values in all four phases of the experiment.

Figure 9-20 presents another representation for maintenance. The light green colored-graphs represent scattering in *CommJ* and light blue colored-graphs represent *AspectJ* respectively. The scattered points in blue, red and green represents CDA, CDO

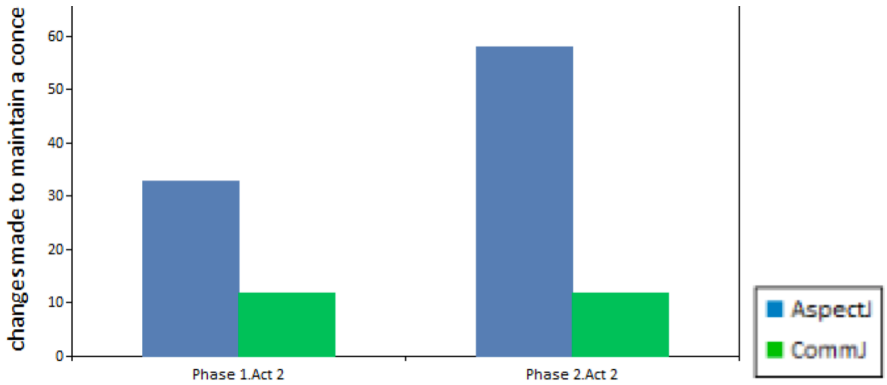

Figure 9-19: CM over Phases

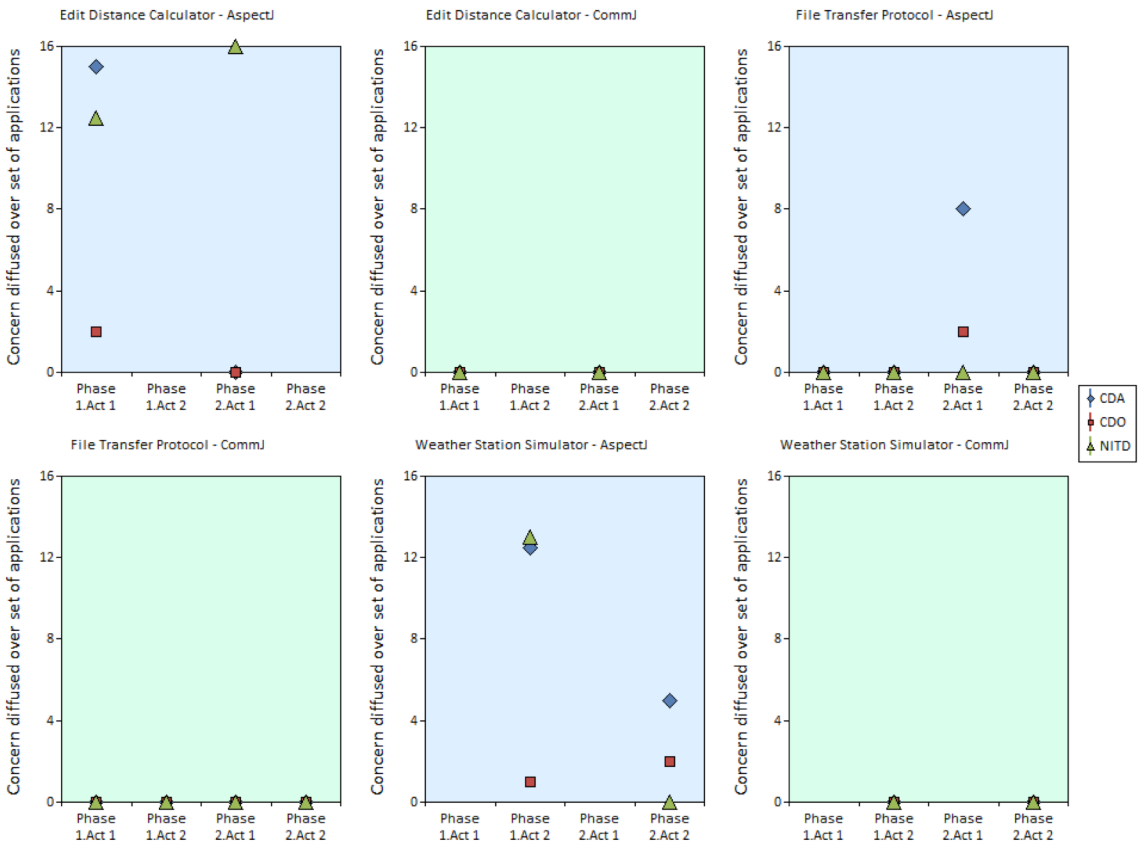

Figure 9-20: CDA, CDO and NITD in *AspectJ* and *CommJ*

and NITD metrics results respectively. The points in graph Figure 9-20 indicate the number of changes for maintaining a program with *CommJ* and *AspectJ* in different activities of Phases 1 and 2, respectively. The results of the graph indicate that CDA, CDO and NITD were zero for all the activities of *CommJ* (highly maintainable) but were highly scattered in *AspectJ*. The reason for reduced values for CDA, CDO and NITD is already discussed in Sections 9.1. and 9.6.

From these results, we can confidentially conclude that Hypothesis#7 holds true for more reusable and maintainable software in *CommJ* than *AspectJ.*

9.8. Other Useful Observations

Besides analysis of the hypotheses, we also collected a handful observations from participants' questionnaires (Appendices D and G) and daily journals during each phase of the experiment.

In regards to understandable code, we found that 100% of *AspectJ* participants in the Phase 1 were confused in identifying pointcuts for implementing the given extension part, and 33% of the same participants were still confused during Phase 2. On the other hand, none of the *CommJ* participants struggled with identifying pointcuts during either phase. This tells us that *CommJ* implementation provides simple pointcuts with understandable IPC abstractions.

For reusability, we observed that 67% of the *AspectJ* participants in Phase 1 agreed that their applications might not run after removing the extension part from the original application. This percentage further increased to 100% in Phase 2. On the other hand, none of the *CommJ* participants made this observation for either phase. This indirectly reconfirms Hypothesis #7, which states that *CommJ* implementations help in developing more reusable crosscutting concerns.

Similarly, for maintainability, 100% of the *AspectJ* participants said that their changes introduced new dependencies in the original sample application after both phases. However, none of the *CommJ* participants felt that they introducedany dependencies during either phase. So, this reconfirms our Hypothesis #7, which asserts that *CommJ* implementation helps in developing more maintainable programs.

The survey also provides information on frequency of bugs. Specifically, 67% of the participants in *AspectJ* group said that their extensions introduced new failures, i.e.,

bugs, into the application code during Phase 1. This percentage further increased to 100% for Phase 2. However, only 25% of the *CommJ* participants in Phase 1 and Phase 2 made this statement. This tells us that *CommJ's* modularization and obliviousness decreased the failures and debugging time.

CHAPTER 10

RELATED WORK

10.1. Work on Communications and Composability with Reference to *CommJ*

We found many papers wherein aspect-oriented technology was for crosscutting concerns related to concurrency and distribution, such as replication [31], persistence [32], synchronization [33], [34], remote pointcuts [35]. However, we did not find any techniques for modularizing crosscutting communication concerns as aspects. To our best knowledge, the closest idea to our research discusses composition of communication abstractions by separating out the definition of communications from the definition of other aspects using general-purpose abstract communication model [36]. We believe our work enables better modularization and obliviousness for IPC concerns.

Marco, et al., describe a *Java* -based communication middleware, called *AspectJRMI* [37] that applies AOP concepts to modularize the design and implementation of *RMI*. Their major contribution is the decomposability of *RMI* into small crosscutting concerns. The idea of horizontal decomposition and defining reusable crosscutting concerns for *RMI* is somewhat similar to *CommJ* design; however, it has a number of differences. First, it is targeting just *RMI*, while our research is more about modeling *IPC* concerns. Second, *CommJ* tries to define a communication joinpoint model, which is not the only contribution of this paper.

We also found some similar ideas of defining reusable communication constructs in *Erlang* language [38], which is based on communication processes using asynchronous message passing. It provides clearly defined communication primitives for *IPC*. In

another paper, they also developed a tool using the above communication abstractions [38] to test concurrent systems. We found some conceptual similarity with this design approach to our work, but their scope in communications is very limited as compare to the *CommJ*.

Gary, et al., describe an approach to build a customized protocol *Cactus* [40], a system in which micro-protocols implement individual attributes of transport that can be combined into a composite protocol that realizes the desired overall functionality. The protocol allows customization of a number of properties, including reliable transmission, congestion detection and control, jitter control, and message ordering. The idea is similar in the sense that *CommJ* allows many reusable aspects, which can be extended to build more useful application-level aspects. In the future, we can define more reusable aspects, which not only can be extended but can also be combined to build more complex types of communication concerns.

Dirk, et al., presents a transformational approach (a Modularized Communication Model) on communication view [41]. The author shows how to separate the definition of communication from the definition of other system aspects, how to extract this definition from existing systems, and how to weave it back into the system. The main concern it tackles is the reconfiguration of communication aspects. Although this paper tries to abstract the communication concerns from core application functionality, it does not talk about the extensions to write reusable, application-level communication aspects as explained in *CommJ*.

A paper on Extensible client-server software by Coady, et al. talks about requiring a clear separation of core services from those that are customizable [42]. This separation

is difficult, as these customizable features tend to crosscut the primary functionality of the core services. The authors sketch out aspects for an *NFS-based client-server architecture* using an *AspectJ* language. However, they talk about handling low-level communications. Although *CommJ* can handle the consistency- and performance-related concerns between initiator and listener, but it describes them in high-level communication abstractions rather than low-level abstractions.

Remi, et al., talk about concurrent event-based AOP [13], which defines the approach of writing concurrent aspects. It first defines a model for concurrent aspects that extends from sequential event-based AOP. Then, it shows how to compose concurrent aspects using a set of general composition operators and sketches its *Java* prototypical implementation. The way the paper tries to compose concurrent aspects shares some similarity with *CommJ*; however, its scope is focused more on concurrency than communications.

Lodewijk, et al., introduces a general model of multi-dimensional concern composition [43] and defines so-called *composition anomalies*. The authors argues that building software by composition of components is far from trivial and fails when components express complex behaviors such as constraints, synchronization and history-sensitiveness. *CommJ* already provides a set of reusable aspects and have the ability to compose using these reusable aspects, but it still needs to consider effects due to composition anomalies.

10.2. Works Related to *CommJ's* Joinpoint Model

Chanwit et al. propose a distributed advice code execution [44]. This interesting idea proposes distributed advice execution using shared execution units. Along the similar lines, Ruben introduces a complete aspect remoting service with one-to-one and one-to-many abstractions, and outlines a distributed joinpoint model to intercept remote services [45]. The notion of remote service abstractions, such as one-to-one and one-to-many abstractions and later its implementation as anypointcut, manypointcut and multipointcut share some design principles with our work.

The main contribution of Muga, et al., in their paper on remote pointcut is to propose a remote pointcut and remote inter-type declaration, an extension to *AspectJ* language for distributed software [35]. The language construct, called remote pointcut, enables developers to write simple aspects to modularize crosscutting concerns related to distributions, scattered on multiple hosts. Similarly, Renaud et al. present a framework to build aspect-oriented distributed applications in *Java* [46]. They discuss dynamic wrappers (also called generic advice) and meta-model annotations to add well-separated concerns. The authors provide a way to define distributed pointcuts. This paper shares some design similarities and future extension points for *CommJ*.

Luis presents three contributions in his paper [36]. First, he introduces a new pointcut language for distributed programming. Second, a notion of distributed advice with support for asynchronous and synchronous executions is defined. Third, he describes distributed aspects including models for deployment, instantiation and state sharing of aspects. These models for deployment, instantiation and state sharing can be another future extension to *CommJ*. His programming patterns proved not so successful

in the distributed environment over irregular communication topologies and heterogeneous synchronization requirements [47]. Luis introduces well-known computation and communication patterns like pipelining, etc., a proposal of language support and their prototypical implementation. *CommJ* design principles include a similar concept for implementing these communication patterns using a language support.

10.3. Work on Interesting Literature with Reference to *CommJ*

Some other authors have explored variousways to deal with inter-concern dependencies between replication and communication [31], [34]. This approach allows reasoning about these inter-dependencies at different levels of abstractions and at the same time discusses the composition of those concerns. Our work focuses primarily on the communication side and is more elaborative. Additionally, we hope that replication concerns composed with communication concerns, programmed using *CommJ* can provide more modular design abstractions.

In his paper, Carlos presents a collection of concurrency patterns using *AspectJ* and compares its benefits with plain *Java* implementation [33]. He presents two alternate implementations: one based on traditional pointcut interfaces and another based on annotations. The aspect-oriented implementation provides high-level reusability, unpluggability, and do not introduce additional overhead when aspects are not included in the build. We believe that in using *CommJ* the same level of concurrency patterns can be redefined in a more modular and oblivious fashion.

The main contribution of the Soargo, et al., is to provide architectural guidelines and implementation of several persistence and distribution concerns in the application

using *AspectJ* [32]. Their purpose is to demonstrate that coding crosscutting concerns using *AspectJ* is a better option than writing in plain *Java* language. This paper shares some architectural guidelines with *CommJ* architecture.

Netinant describes an aspect-oriented framework wherein both functional components and system properties are designed relatively separate from each other [48]. This separation of concerns allows for reusability and enables the building of software systems that are manageable, stable and adaptable. Most of the work in Netinant's paper concentrates on the decomposition of concurrent object-oriented systems with the goal to achieve an improved separation of concerns in both design and implementation. It highlights the general design principles for separation of concerns, some of which can be employed in *CommJ* to improve its existing design.

10.4. Work on Measurement Metrics with Reference to EQM

McCall identifies a list of 11 quality attributes that have an important influence on quality of the software [24]. In our experiment's perspective, we decided that maintainability and reusability would be the most important.

We use Sant'Anna's quality model [10] because it is more generalized to measure different concerns of design and code as compared to Lopes' work [2]. Additionally, Sant'Anna's is strong enough to be applied to all three different types of implementations. Some other metrics [49] can be considered as complimentary to our chosen quality model, but they are not based on well-known software engineering quality models.

Sant' Anna builds the Quality model [10] using Basili's *GQM Methodology* [50]. Basili provides a three-step framework: (1) list the major goals of the empirical study, (2) derive from each goal the questions that must be answered to determine if the goals have been met; (3) decide what must be measured in order to be able to answer the questions adequately.

We also made a few enhancements to the quality of the model [10] and hope that doing so would further strengthen the model. For instance, interpretation of maintainability and reusability is dependent upon flexibility and understandability factors. As per our definitions of qualities (Section 6.2), code obliviousness [44] and localization of design decisions [27] are two very important missing factors in the model. Research and practices also validate that modular code is more maintainable [12]. Further, Parnas previously defined three properties of modular code as being flexibility, comprehensibility and independent development [27]. At that time, code obliviousness was not the primary concern but became an important element of software design in later years after emerging research in *AOSD*.

Because our research method is of an empirical nature and depends on a quality model [10], our model is neither a fully qualitative or quantitative but a combination of both. Some parts of the model are quantitative, such as quality metrics, but others, such as qualities, factors, and internal attributes, are of qualitative nature, and rely on an inductive processes.

CHAPTER 11

SUMMARY AND FUTURE WORK

11.1. Summary

Our research introduces the notation of communication and connection aspects and discusses an *AspectJ* framework, namely *CommJ*, for weaving aspects into IPC. It then describes the design and implementation of some of *CommJ* components, such as the base aspects. It also provides an overview of a toolkit, i.e., the RAL that consists of reusable communication aspects and doubles as a proof of concept, since these aspects can be directly applied to a wide range of existing applications. We believe that *CommJ* is capable of encapsulating a wide range of communication-related and connection-related crosscutting concerns in aspects. We hope to gather more empirical evidence of the *CommJ's* value by increasing the number of aspects in the RAL and by continuing to expand the number and types of applications that use *CommJ*. We also conducted a research experiment to compare *AspectJ* with *CommJ* for various software design attributes related to reuse and maintenance through an extended quality model. Initial findings from this experiment revealed that crosscutting concerns programmed in *CommJ* delivered more modular, reusable and maintainable programs. However, our future research will include more formal software-engineering productivity experiments to verify this belief.

11.2. Future Work

We envision a number of extensions or spins off to *CommJ*. First, distributed transaction processing systems is another high-level programming concept that can be unnecessarily complex when crosscutting concerns, e.g. logging, concurrency controls, transaction management, and access controls, are scattered throughout the transaction processing logic or tangled into otherwise cohesive modules. We can use the same approach that we used for *CommJ* to extended *AspectJ* for the weaving of crosscutting concerns in transactions.

Second, *CommJ* can also be extended for distributed pointcuts that would simplify the implementation of even more complex crosscutting concerns, such as object-replication, migration, or fragmentation in a distributed system.

Finally, *CommJ* has the potential to be very useful for testing various kinds of time-sensitive communication related errors in IPC. We plan to explore this potential and additional experiments focus on quality of service and timing issues related to IPC.

REFERENCES

[1] G. Kiczales, G. *et* al., "Aspect-oriented programming," In *Proc. 11th ECOOP*, Jyvaskyla, Finland, 1997, pp. 220-242.

[2] C. Lopes, "D: A language framework for distributed programming," PhD. dissertation, Coll. Comp. Sci., Northeastern University, Boston, MA, 1997.

[3] Eclipse, 2014, Sep 13, AspectJ, [Online]. Available: http://www.eclipse.org/aspectJ/

[4] AspectWorkz2, 2005, Plain Java AOP [Online]. Available: http://aspectwerkz.codehaus.org/

[5] JBoss, 2014, Sep 13, JBoss AOP, [Online]. Available: http://www.jboss.org/jbossaop

[6] Spring Framework, 2014, Sep 13, Spring AOP, [Online]. Available: http://www.tutorialspoint.com/spring/aop_with_spring.htm

[7] C. Clifton and G T. Leavens, "Obliviousness, modular reasoning, and the behavior subtyping analogy," In *Proc. 2nd Int. Conf. AOSD SPLAT Workshop*, Boston, MA, 2003, pp. 1-6.

[8] L. Bergmans , *et al.*, "Composing software from multiple concerns: Composability and composition anomalies," In *ICSE Workshop on Multi-Dimensional Separation of Concerns in Software Engineering*, Limerick, Ireland, 2000.

[9] Office of Research and Graduate Studies at Utah State University, (2014, May 26) *Institutional Review Board* [Online]. Available: http://rgs.usu.edu/irb/

[10] C. Sant'Anna *et al*., "On the reuse and maintenance of Aspect-Oriented Software: An assessment framework," In *Proc. 17th Brazilian Symp. Software Engineering*, Manaus, Brazil, 2003, doi: PUC-RioInf, MCC26/03.

[11] G. Gamma *et al*., *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley, 1995.

[12] G. Kiczales and M. Mezini, "Aspect-oriented programming and modular reasoning," In *Proc. 27th Int. Conf. Software Engineering*, St. Louis, MO, 2005, pp. 49-58.

[13] R. Douence *et al.*, "Concurrent aspects," In. *Proc. 5th Int. Conf. GPCE*, Portland, OR, 2006, pp. 79-88.

[14] W. De Meuter, Monads as a theoretical foundation for AOP, In *Int. Workshop on AOP at 11th ECOOP*, 1997, Springer-Verlag. doi: 10.1.1.2.4757

[15] P. Tarr *et al*., "N degrees of separation: Multi-dimensional separation of concerns," In *Proc.21st Int. Conf. Software Engineering*, Los Angeles, CA, 1999, pp. 107-119.

[16] H. Ossher and P.Tarr. "Multi-dimensional separation of concerns and the hyperspace approach," IBM, Yorktown Heights, NY, IBM Res. Rep. 21452, April, 1999.

[17] W. Harrison and H. Ossher, "Subject-oriented programming - A critique of pure objects," In *Proc. 8th Conf. on Object-Oriented Programming Systems, Languages, and Applications*, Oakland, CA,1993, pp. 411-428.

[18] S. Chiba. "Load-time structural reflection in Java," In *Proc.14th ECOOP*, Cannes, France, 2000, pp. 313-336.=

[19] T. J. Brown *et al*., " Mixin programming in Java with reflection and dynamic invocation," In *Proc.of the Inaugural Conf. on the Principles and Practice of programming, and Proc. 2nd Workshop on Intermediate Representation Engineering for Virtual Machines*, Dublin, Ireland, 2002, pp. 25-34.

[20] Wikipedia. 2013, Feb. 09. *Block Diagram*. [Online]. Available: http://en. wikipedia.org/wiki/Block_diagram

[21] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*., Upper Saddle River, NJ: Prentice-Hall, 1996.

[22] G. Kiczales and M. Mezini. "Aspect-oriented programming and modular reasoning," In *Proc. 27th Int.Conf. Software Engineering*, St. Louis, MO, 2005, pp. 49-58.

[23] S.R. Chidamber and C. F. Kemerer, "A metrics suite for object-oriented design," *IEEE Trans. Softw. Eng*., vol. SE-20, no. 6, pp. 476–493, June 1994.

[24] J.A. McCall *et al*., "Factors in software quality," NTIS, Alexandria, VA, Tech. Rep. AD-A049-014, 015, 055, 1977.

[25] *IEEE Standard for Software Maintenance*, IEEE Standard 1219-1998, 1998.

[26] R.E. Filman and D. P. Friedman, "Aspect-oriented programming is quantification and obliviousness," IEEE RIACS Tech. Rep. 01.12, May 2001.

[27] D. L. Parnas. "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no.12, pp. 1053-1058, Dec. 1972.

[28] T.J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng*., vol. 2, no. 4, pp. 308-320, Dec. 1976.

[29] SourceForge, 2014, Sep 13, *Eclipse Metrics Project 1.3.6* [Online]. Available: http://metrics.sourceforge.net

[30] E. Figueiredo *et al*., "AJATO: An AspectJ assessment tool," In *Proc. 20th ECOOP*, Nantes, France, 2006.

[31] M. Nishizawa and S. Chiba, "Jarcler: Aspect-oriented middleware for distributed software in Java, Tokyo Inst. of Tech., Tokyo, Japan, Dept. of Math. And Comp. Sciences Res. Rep. C-164, 2002.

[32] S. Soares *et al.*, "Implementing distribution and persistence aspects with AspectJ," In *Proc. 17th ACM SIGPLAN Conf. OOPSLA*, Pittsburgh, PA, 2002, pp. 174-190.

[33] C. A. Cunha *et al.*, "Reusable aspect-oriented implementations of concurrency patterns and mechanisms," In *Proc. 5th Int. Conf. AOSD*, Bonn Germany, 2006, pp. 134-145.

[34] M. Antunes *et al,*, "Separating replication from distributed communication: Problems and solutions," In *2001 Int. Conf. on Distributed Computing Systems Workshop*, Mesa, AZ, 2001, pp. 103-108.

[35] M. Nishizawa *et al.*, "Remote pointcut – A language construct for distributed AOP," In *Proc. 3rd Int. Conf. AOSD*, Lancaster, UK, 2004, pp. 7-15.

[36] L.D. Benavides Navarro *et al.*, "Explicitly distributed AOP using AWED," In *Proc. 5th Int. Conf. AOSD*, Bonn Germany, 2006, pp. 51-62.

[37] M.T. de Oliveira Valente *et al*., 2005. "An aspect-oriented communication middleware system," In *Proc. 2005 OTM Confederated Int. Conf. On the Move to Meaningful Internet Systems: CoopIS, COA, and ODBASE - Volume Part II*, Agia Napa, Cyprus, 2005, pp. 1115-1132.

[38] M. Christakis and K. Sagonas, "Detection of asynchronous message passing using static analysis," In *Proc.13th Int. Conf. Practical Aspects of Declarative Languages*, Austin, TX, 2011, pp. 5-18.

[39] Gregor Kiczales and Mira Mezini. 2005. Aspect-oriented programming and modular reasoning. InProceedings of the 27th international conference on Software engineering (ICSE '05).

[40] G. T. Wong *et al*., "A configurable and extensible transport protocol," *IEEE/ACM TON*, vol. 15, no. 6, pp. 1254-1265, Dec. 2007.

[41] D. Heuzeroth *et al.,* "Aspect-oriented configuration and adaptation of component communication," in *Proc. of 3rd Int. Conf. on GCSE*, Erfurt, Germany, 2001, pp. 58-69.

[42] Y. Coady *et al.*, "Can AOP support extensibility in client-server architectures?" in *Proc. ECOOP Aspect-Oriented Programming Workshop*, Budapest, Hungary, 2001.

[43] L. Bergmans and M. Aksit, "Composing software from multiple concerns: A model and composition anomalies," In *ICSE Workshop on Multi-Dimensional Separation of Concerns in Software Engineering*, Limerick, Ireland, 2000.

[44] C. Kaewkasi and J. R. Gurd, "A distributed dynamic aspect machine for scientific software development," In *Proc.1st Workshop on VMIL*, 2007, ACM. doi: 10.1145/1230136.1230139

[45] R. Mondejar *et al.*, "Building a distributed AOP middleware for large scale systems," In *Proc. 2008 Workshop NAOMI*, Brussels, Belgium, 2008, pp. 17-22.

[46] R. Pawlak *et al.*, "JAC: An aspect-based distributed dynamic framework," *J. Software, Practices and Experience*, vol. 34, no.12, pp. 1119-1148, Oct., 2004.[47] L.D. Benavides Navarro *et al.*, "Invasive patterns for distributed programs," In *OTM Confederated Int. Conf*., Vilamoura, Portugal, 2007, pp. 772-789.

[48] P. Netinant, "Composition of system properties," Bankok University, Tech. Rep., 2004.[49] J. Zhao "Towards a metrics suite for aspect-oriented software," Info. Processing Soc. of Japan, Tokyo, Japan,Tech. Rep. SE-136-25, Mar. 2002.

[49] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming language. In Research Directions in Concurrent Object-Oriented Programming, pages 107–150. 1993.

[50] V. Basili *et al*., "The goal question metric approach," In *Encyclopedia of Software Engineering*, vol. 2, J.J. Marciniak, Ed. Hoboken, NJ: Wiley, 1994, pp. 528-532.

[51] Raza A., Clyde S., Weaving Crosscutting Concerns into Inter-Process Communication (IPC) in *AspectJ*. In ICSEA 2013, Venice, Italy, pp. 234-240.

[52] Raza A., Clyde S., Communication Aspects with *CommJ*: Initial Experiment Show Promising Improvements in Reusability and Maintainability. In ICSEA 2014, Nice, France (Submitted).

[53] Deuence R., Motelet O., Sudholt M., A formal definition of crosscut, MISC 2001.

[54] Crista L., D: A Language Framework for Distributed Programming Ph.D. Thesis. College of Computer Science, Northeastern University, Dec 1997 (1998).

[55] E. Frchi *et al.*, "Concurrent bug patterns and how to test them," In *Int. Symp. On Parallel and Distributed Processing*, Nice, France, 2003, p. 286.2.

APPENDICES

APPENDIX A

SELECTED SAMPLE APPLICATIONS

A.1. Levenshtein Edit-Distance Calculator

This system allows users to enter two words into the client console, which then requests a server to compute the *Levenshtein Distance*, LD, between the two words, wherein LD is the minimum number of single-character edits (insertion, deletion, and substitution) required to change one word into the other. For example, the LD between "kitten" and "sitting" is 3, since the following three edits change one into the other, and there is no way to do it with fewer than three edits:

- **k**itten → **s**itten (substitution of "s" for "k")
- sitt**e**n → sitt**i**n (substitution of "i" for "e")
- sittin → sittin**g** (insertion of "g" at the end)

Figure A-1 shows an overview of the current architecture for this system. It only contains three classes, Client, Calculator, and Message. Both the Client and Calculator run as separate processes, and may even be on separate machines. The Client allows the users to type in two words using a simple console interface. Then, it creates an instance of the Message class containing these two words and sends it to the calculator. The calculator computes the LD and a package that result in a new instance of Message, and sends it back to Client. The UML Sequence Diagram in Figure A-2 shows this interaction.

Note, that the interaction is asynchronous from the Client's perspective. In other words, the Client does not block while waiting for a response to the translation request.

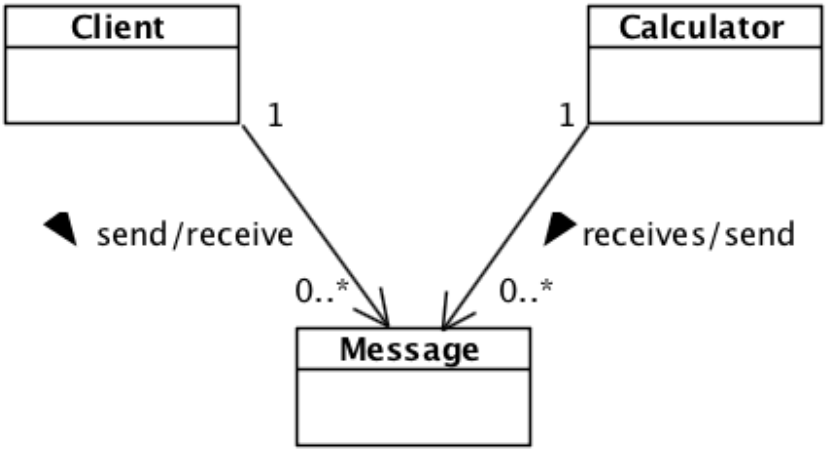

Figure A-1: Architecture Diagram of Levenshtein Edit-Distance Calculator

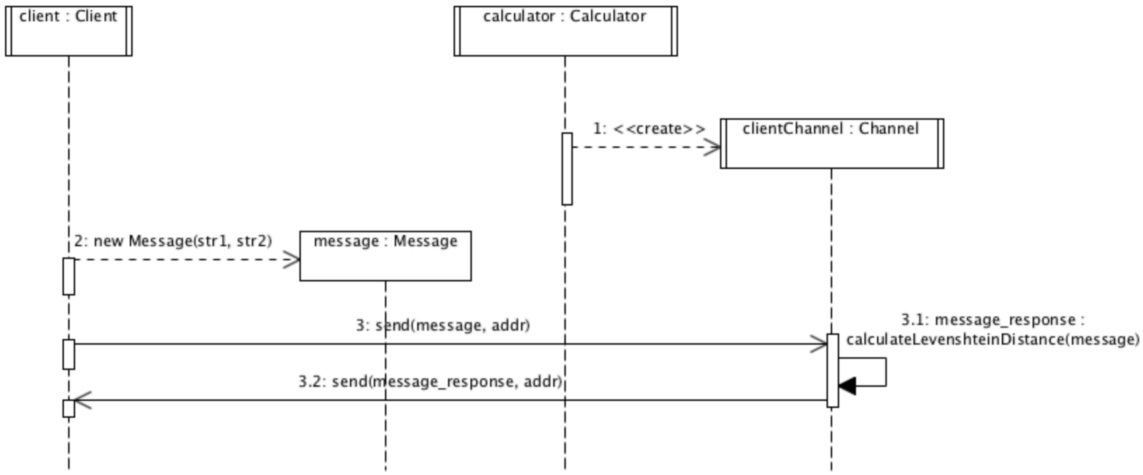

Figure A-2: Interaction Diagram between Client and Edit-Distance Calculator

A.2. File Transfer Protocol

FTPClient requests *FTPServer* for a list of available files and then sends a file download request to the server. The server sends the requested file in small chunks to the client.

Figure A-3 shows an overview of the current architecture for this system. It only contains two main classes, i.e., *FTPClient*, *FTPServer* and three protocol messages *FileTransferRequet*, *FileTransferResponse* and *FileTransferAck*. Both the client and server run as separate processes, and may even be on separate machines. The UML Sequence Diagram in Figure A-4 shows this client-server interaction in more detail.

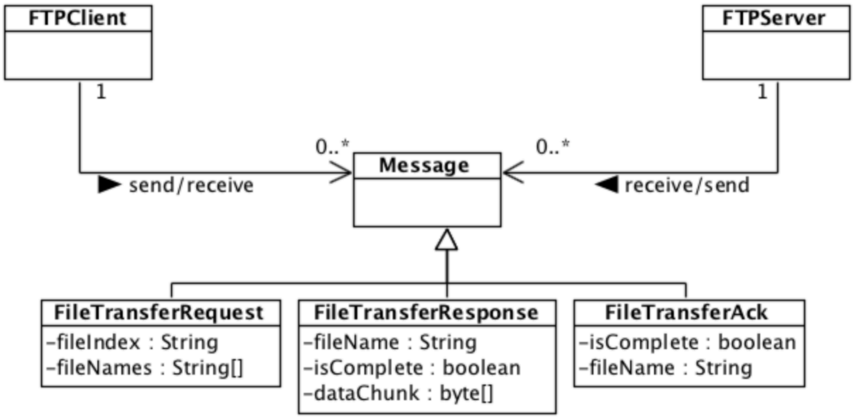

Figure A-3: Architecture Diagram for FTP

FTPClient communicates with the *FTPServer* and establishes a TCP connection. The client sends a FileTransferRequest to the server to ask for the list of available files on the server. *FTPServer* sends back the list of available file names, encapsulated in FileTransferRequest. *FTPClient* then allows the user to enter the selected file index, using console input. Then it creates an instance of FileTransferRequest, encapsulated

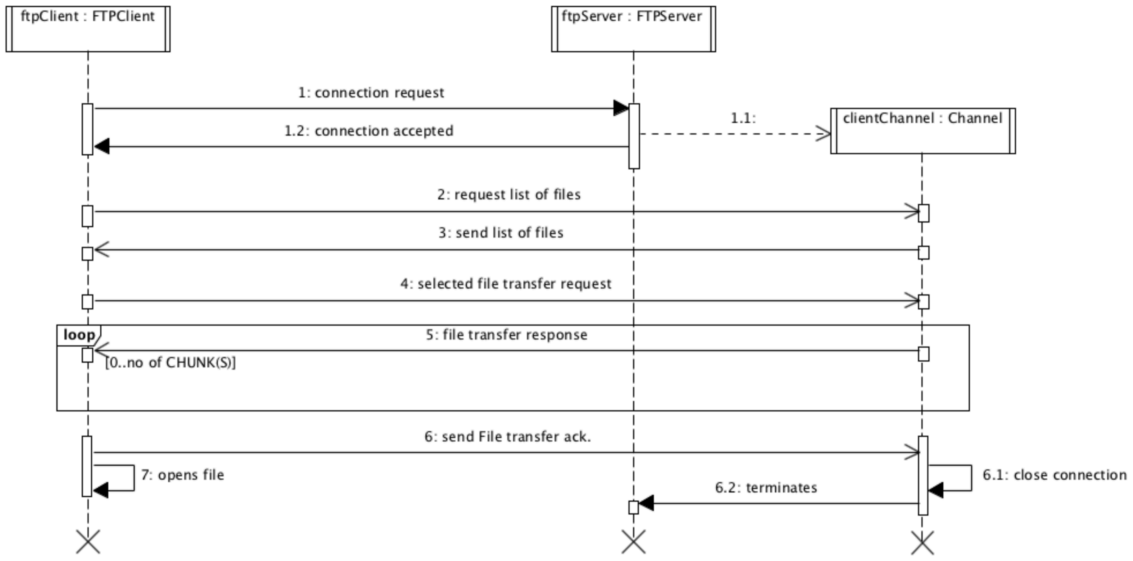

Figure A-4: Interaction Diagram between *FTPClient* and *FTPServer*

with selected file index, and sends to the server. *FTPServer* receives the request, and starts transferring the selected file contents in fixed-length data chunks, encapsulated in FileTransferResponse. Once the file has been successfully transferred, client sends an acknowledgement message, *FileTransferAck*, to the *FTPServer*. *FTPClient* process automatically opens the file after successful transfer and terminates itself. *FTPServer* terminates itself after the file has been transferred successfully and has received an acknowledgement.

Note, that the interaction is asynchronous from both the client and server perspective. In other words, both the client and server does not block while waiting for a protocol message.

A.3. Weather Station Simulator

This example simulates a typical weather station consisting of three main components, i.e., *WeatherStationSensor, Transmitter* and a *Receiver.*

WeatherStationSensor runs in a thread, generates weather-data readings at random intervals and temporarily stores them in a queue, accessible to the *Transmitter*. On receiving a request weather-data from the *Receiver* in random intervals, the *Transmitter* sends all of the data available in the queue, one weather-data reading at a time and in order, to *Receiver*. *Receiver* periodically sends more requests for weather data if it has not received any data for some time period.

A.3.1. Current Design

Figure A-5 shows an overview of the current architecture for WeatherStationSimulator and protocol messages. The system contains three main classes ,i.e., *WeatherStationSensor, Transmitter* and *Receiver. WeatherStationSensor* generates WeatherDataVector(s) (weather-sensitive observations). *Transmitter* collects WeatherDataVector(s) and sends them to the *Receiver.* Figure A-6 describes the *WeatherStationSensor* design. The UML Sequence Diagram in Figure A-7 shows the *Transmitter* /*Receiver* interactions in more details.

Application runs two instances of *Transmitter* and one instance of *Receiver.* Each *Transmitter* starts its own *WeatherStationSensor* thread. The sensor combines the readings from its various sub-components (Figure A-6) into a WeatherDataReading object. It then generates an instance of WeatherDataVector message, and populates it with four WeatherDataReading instances, at random intervals, and stores in a temporary data structure. Finally, a glossary is provided at the end of this appendix.

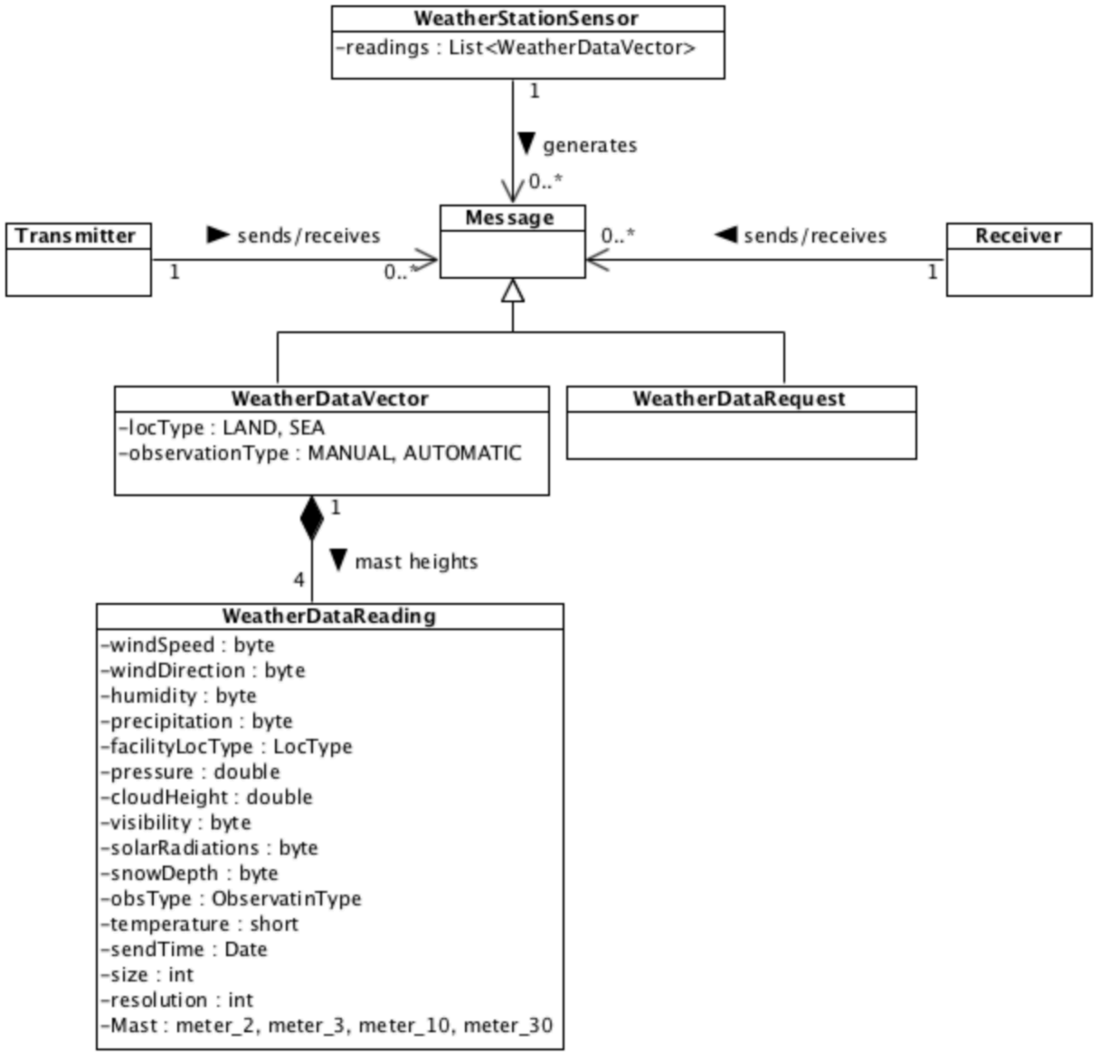

Figure A-5: Data Structures for Weather Station Simulator Example

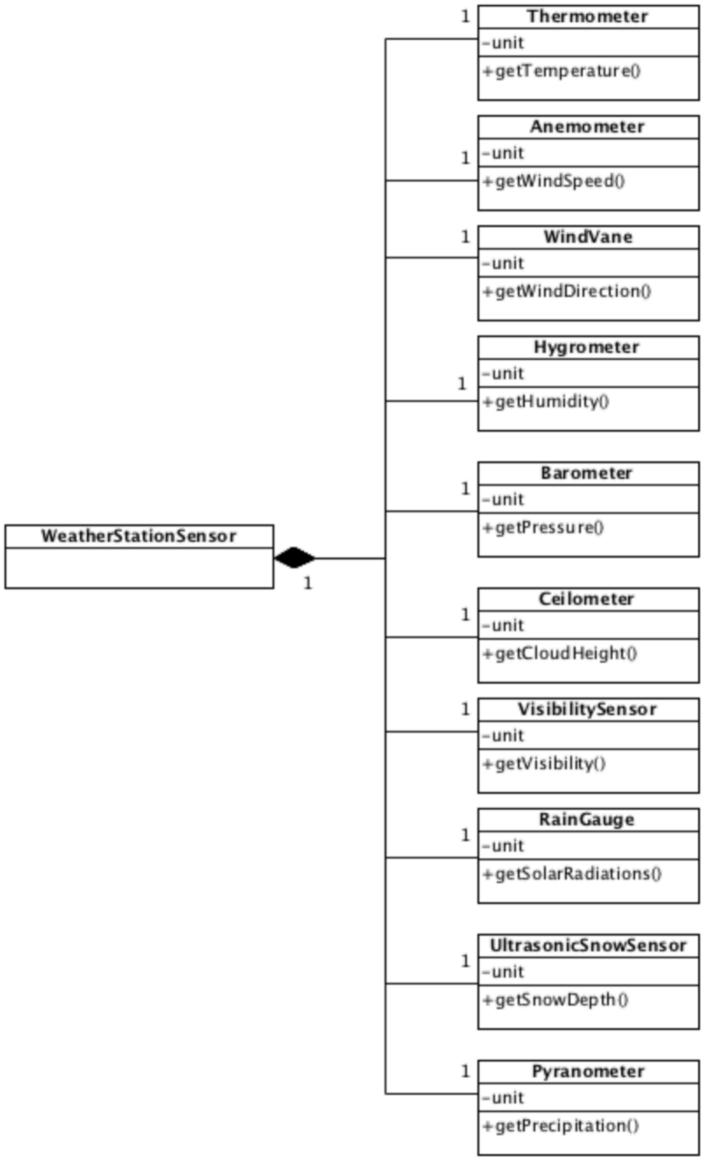

Figure A-6: Weather Station Simulator

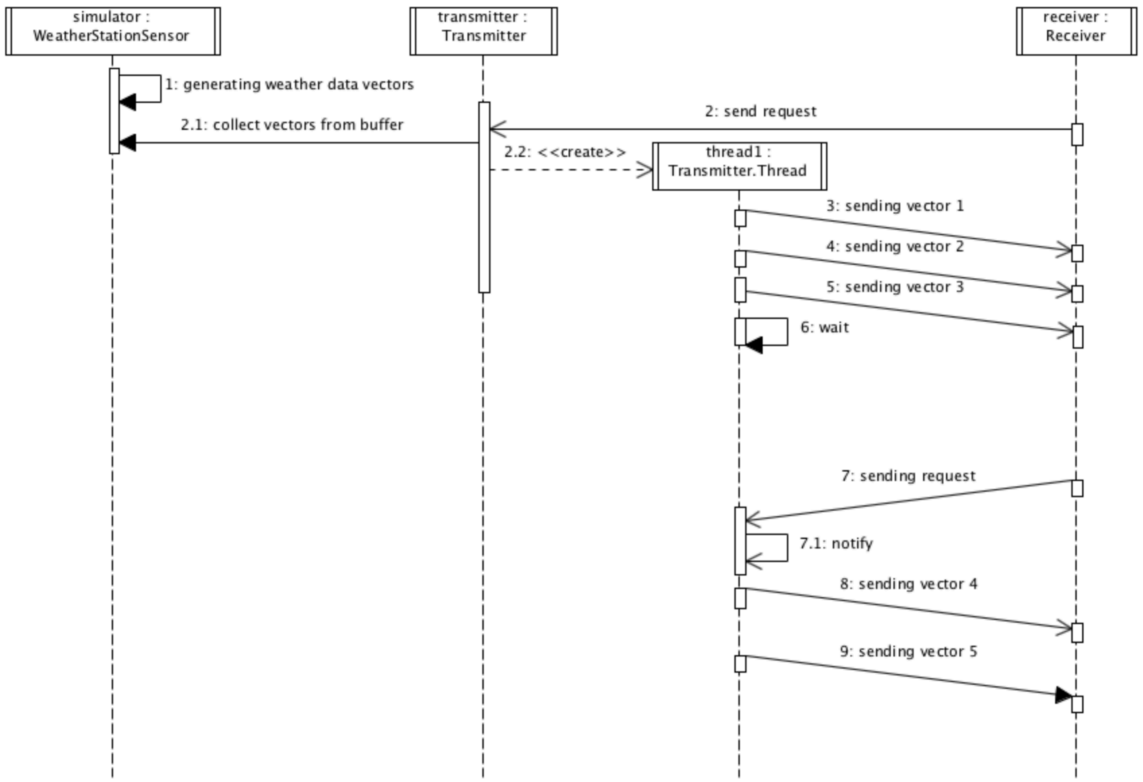

Figure A-7: Interaction Diagram between *Transmitter* and *Receiver*

GLOSSARY	
Term	**Description**
Thermometer	It is used for measuring temperature
Anemometer	It is used *for measuring wind speed*
Wind vane	It is used for measuring wind direction
Hygrometer	It is used for measuring humidity
Barometer	It is used for measuring atmospheric pressure
Ceilometer	It is used for measuring cloud height
Visibility sensor	It is used for measuring visibility
Rain gauge	It is used for measuring liquid-equivalent precipitation
Ultrasonic snow sensor	It is used for measuring depth of snow
Pyranometer	It is used for measuring solar radiations
Mast Heights	A pole, or long, strong, round piece of timber, or spar, set upright in a boat or vessel to note weather readings

APPENDIX B

SELECTED INTER-PROCESS EXTENSIONS

B.1. Version Compatibility

This extension adapts one version of the message to another, so processes running different versions can still communicate with each other. In addition:

- Each application process knows its version number.
- Each message contains that version number.
- Before sending the message to *Receiver,* this extension always converts the message to its application version on the sender side.
- After receiving the message, it always ensures that the received message is matched with the application version at *Receiver* side.

B.2. Measuring Performance

This extension measures some performance-related statistics for message-based communications between a *Sender* and *Receiver.* In addition, the extension logs the following performance related statistics:

- Total numbers of conversations, which occurred in the system where a conversation can be defined with any combinations of, sends or receives. Different types of conversations are one-way send, one-way receive, request-reply and multi-step conversations
- Total time for all conversations
- Average turnaround time for a request to be processed where average turn-around time is the average of a timespan from conversation start time to conversation end time
- Maximum turnaround time for any conversations

- Minimum turnaround time for any conversation
- The program logs the time when a conversation starts
- It logs and calculates the above statistics when the conversation ends
- Note that a conversation can be a simple request-reply type exchange of messages or a complex combination of send and receive events. We define the conversations for sample applications as follows:
    - *Levenshtein Edit-Distance Calculator:* A conversation is when a client sends a request and receives a response from the calculator
    - *File Transfer Protocol:* A conversation is when a client sends a request for a file download and when it receives the last response of data chunk for that file from the server
    - *Weather Station Simulator:* A conversation is when a *Receiver* sends a request to get weather-related data readings and receives the first response from the *Transmitter*
- Developers would be provided with the following classes:
- *Stats:* A data structure containing elements to measure performance
- *PerformanceMeasure:* It logs performance measure using sliding window

B.3. Symmetric-Key Encryption

The program encrypts the communication between a sender and receiver using symmetric-key encryption. In addition to that:

Exchanging secret keys

- The program first starts a *KeyManager* process, which handles the key requests from *Sender* and *Receiver* processes. We assume that both the *Sender* and *Receiver* are already registered with the *KeyManager*.
- *Sender* starts a *KeyClient* process, which sends a *KeyRequest* message to *KeyManager*. The *KeyManager* authenticates the sender, creates a *SharedKey*, encapsulates it in *KeyResponse* message, and sends it to *Sender*.
- *Receiver* also creates a *KeyClient*, which sends a *KeyRequest* to *KeyManager*. The *KeyManager* again authenticates the *Receiver*, creates a *SharedKey*, encapsulates it in *KeyResponse* message, and sends it to *Receiver*.
- If *KeyManager* cannot authenticate any processes, it sends an empty *KeyResponse* and the respective process terminates itself on receiving null Key.
- Figures B-1 and B-2 describes the process of exchanging secret keys.
- Message Communications between Sender and Receiver
- Before sending a protocol message, Sender encrypts the message with *SharedKey*.
- After receiving the message, *Receiver* decrypts the Message with *SharedKey*.
- Developers would be provided with the following classes:
- *Encryption:* A data structure containing elements to measure performance.
- *KeyManager:* It authenticates the processes and provides the shared key.

- *KMClient:* It sends the authentication information to *KeyManager* and requests the shared key.
- *KeyRequest:* A protocol message used to request *SharedKey*.
- *KeyResponse:* A protocol message used by *KeyManager* to send *SharedKey*.
- *SharedKey:* This class encapsulates the shared key information.

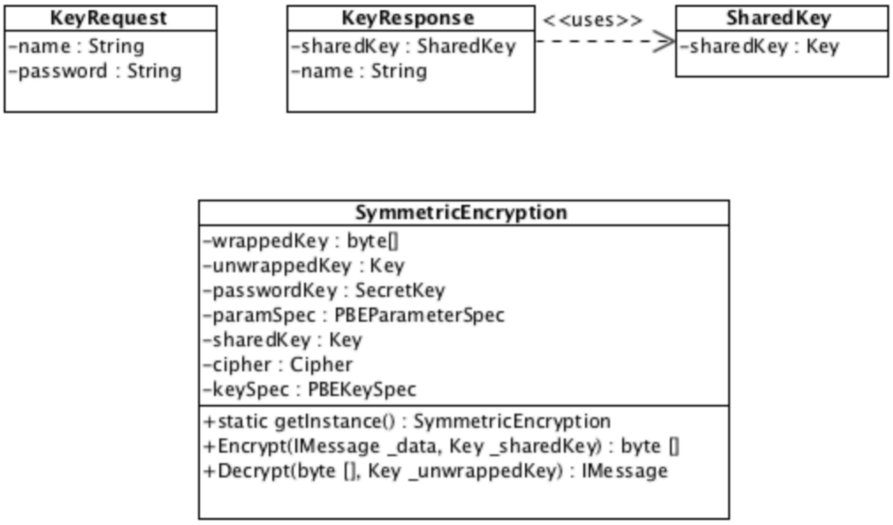

Figure B-1: Data Structures for Symmetric-Key Encryption

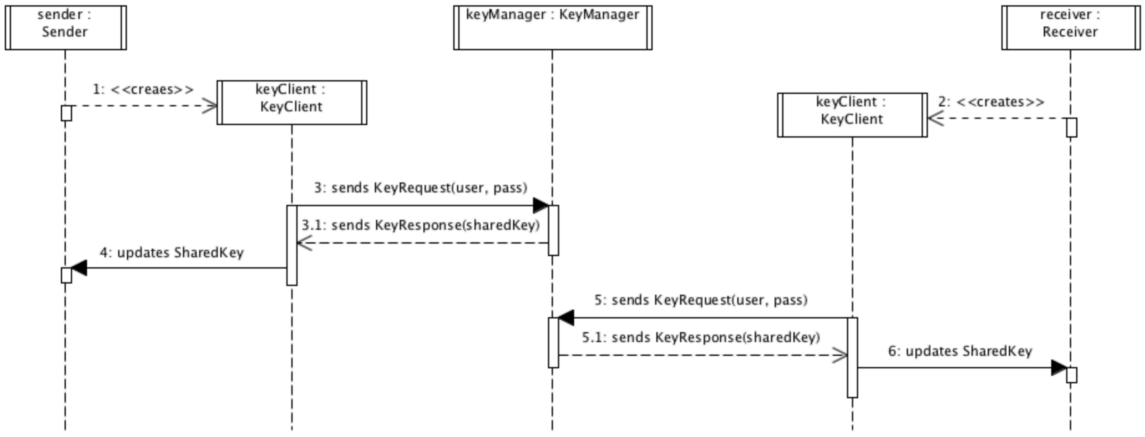

Figure B-2: Process of Exchanging Shared Keys

APPENDIX C

SKILL ASSESSMENT SURVEY

Volunteer # _____

Rank the following on scale from 1-5, where 1 represents beginner, 3 novices, and 5 experts.

1. Beginners will have a working knowledge of the skill, but no practical experience. A novice will have at least 2 year of practical experience, in either academic or industrial settings. An expert will have more than 3 years of experience.
   a. *Java* network programming using channels? 1 2 3 4 5
   b. *Java* network programming using sockets? 1 2 3 4 5
   c. UML? 1 2 3 4 5
   d. Good design principles such as modularity etc. 1 2 3 4 5
   e. Multithreaded programming using *Java?* 1 2 3 4 5

2. Can you quantify in terms of Lines of Code (LoC) for your most complex *Java* programming project?
   a. Less than 1,000 LoC
   b. Between 1,000 and 10,000 LoC
   c. Between 10,000 and 20,000 LoC
   d. Between 20,000 and 100,000 LoC
   e. More than 100,000 LoC

3. How many years of programming experience do you have?
    a. No prior programming experience
    b. Less than 1 year
    c. Between 1-3 years
    d. Between 3-5 years
    e. More than 5 years

4. How many years of *Java* programming experience do you have?
    a. No prior programming experience
    b. Less than 1 year
    c. Between 1-3 years
    d. Between 3-5 years
    e. More than 5 years

5. Please select your favorite programming languages?
    a. *Java*
    b. C#
    c. PHP
    d. Ruby and Rails
    e. C++
    f. Other

6. What is your computer science education background?
    a. BS/BE
    b. MS
    c. Ph.D.
    d. Other

7. Which of the following courses have you taken as part of your computer science curricula?
    a. Object Oriented Design
    b. Software Engineering
    c. Unified Modeling Language
    d. Object-Oriented Programming
    e. Multithreaded Programming
    f. Network/Distributed Programming

APPENDIX D

QUESTIONNAIRE FOR PHASE 1 IMPLEMENTATION

Volunteer # _____

D.1. Phase 1 pre-implementation questionnaire

1. From scale 1-5, how would you rank the existing applications for code tangling (1 means fully tangled and 5 means two are totally independent)?

2. From scale 1-5, how would you rank the existing applications for code scattering (1 means fully scattered in all classes and 5 means no scattering)?

3. If the original application (such as Edit-Distance Calculator and FTP) were implemented using connection-less communications, would your changes have been?
   a. Considerably different
   b. Somewhat different
   c. A little different
   d. No different

4. Now if you were asked to change the implementations (such as Edit-Distance Calculator and FTP) for Phase 1 to connection-oriented communications, would this be?
    a. Major change
    b. Minor change
    c. No different

5. If the original application of WeatherStationSimulator were implemented using connect-oriented communications, would your changes have been?
    a. Considerably different
    b. Somewhat different
    c. A little different
    d. No different

6. Now if you were asked to change the implementation for WeatherStationSimulator in Phase 1 to connection-less communications, would this be?
    a. Major change
    b. Minor change
    c. No different

7. If the original application (such as Edit-Distance Calculator and FTP) were implemented using JDK Sockets rather than JDK Channels, would your changes have been?
    f. Considerably different
    g. Somewhat different
    h. A little different
    i. No different

8. Now if you were asked to change the implementation for original application (such as Edit-Distance Calculator and FTP) back to JDK Channels, would this be?
    a. Major change
    b. Minor change
    c. No different
    d. Considerably different

9. If the original application of WeatherStationSimulator where implemented in such a way so that the *Transmitter* s in the original application, send data readings to multiple *Receiver* s, would your changes be?
    a. Considerably different
    b. Somewhat different
    c. A little different
    d. No different

10. Now if you were asked to change the implementation for WeatherStationSimulator back to the original application where *Transmitter* s are sending the data readings to just one *Receiver,* would this change be?
    a. Major change
    b. Minor change
    c. No different

11. Suppose we want to implement the "Performance Measurement" feature for the original applications. The feature measures some performance related statistics such as turn-around time for message-based communications between a sender and *Receiver.* To implement this feature would your changes be?
    a. Considerably different
    b. Somewhat different
    c. A little different
    d. No different

12. Now suppose if we change the requirements for "Performance Measurement" feature such that a conversation is not only a request-reply sequence but also a request-reply-acknowledgement (multi-step conversation), would this change be?
    a. Major change
    b. Minor change
    c. No different

D.2. Phase 1 post-implementation questionnaire

Volunteer # _____

1. While implementing the initial version of changes for sample applications, which of the following did you find the most difficult?
    a. Adding additional requirements for the extension part to applications design
    b. Deciding how to share data between previously existing sample application code and new code
    c. Debugging the applications with crosscutting concerns
    d. Working with the *Java* implementation language or the IDE
    e. Managing the complexity of the application

2. Which of the following was the most time consuming activity during Phase 1?
    a. Understanding the original applications and analyze the new requirements
    b. Designing the solutions
    c. Implementing the solutions
    d. Debugging the solutions
    e. Learning the tools (e.g., *Java*, an IDE)
    f. Learning AOP (not applicable for group 1)
    g. Learning *CommJ* (not applicable groups 1 and 2)

3. While implementing your changes, did your come across any of the following situations? (Select all that apply)
    a. Your changes introduced new bugs

b. Your changes introduced new dependency among existing application components
c. Tangling and scattering increased
d. None of the above

4. If you were asked to refactor the changes related to the extension part so it could be reused by other applications, which of following would you do?
   a. Redesign the application's structure, making major changes in the classes, their relationships, and responsibilities
   b. Refactor the code to make minor improvements to the classes, their relationships, or responsibilities
   c. Improve the implementation of individual methods, independent of changing the structure of the application, to improve readability or maintainability
   d. Nothing – the implementation is ready for reuse

5. How would you rank your application, so that it would work again if you separate the extension related code files in Phase1 from sample application code?
   a. Very easy change, the two parts are almost oblivious
   b. A little difficult as there are some extension related references exists in the original application
   c. A significant effort is required as some extension related code snippets is tangled and scattered in the original application code or vice versa

6. Suppose your original application (such as Edit-Distance Calculator and FTP) were implemented using connectionless communications. To implement this feature would your changes be?
    a. Considerably different
    b. Somewhat different
    c. A little different
    d. No different

7. If the original application of WeatherStationSimulator where implemented in such a way so that the *Transmitter* s in the original application, send data readings to multiple *Receiver* s. To implement this feature would your changes be?
    a. Considerably different
    b. Somewhat different
    c. A little different
    d. No different

8. If the original application (such as Edit-Distance Calculator and FTP) were implemented using JDK Sockets rather than JDK Channels. To implement this feature would your changes be?
    a. Considerably different
    b. Somewhat different
    c. A little different

d. No different

9. To implement the “Performance Measurement” feature, what are the following changes you made in your original application?
   a. Need to introduce major changes in the original application code
   b. Need to introduce new pointcuts
   c. Need to define new data structures to keep track of conversation
   d. Lines of Code (LoC) and complexity of sample application may increase
   e. Tangling and Scattering of sample application may increase
   f. Require only minor change in implementation
   g. Only need to modify some rules i.e., state machines etc., to accommodate new conversations
   h. May expect some new bugs in the program
   i. Overall debugging time would dramatically increase
   j. Can reuse existing code to implement new changes

10. Suppose if we change the requirements for “Performance Measurement” feature such that a conversation is not only request-reply sequence but also a request-reply-acknowledgement (multi-step conversation), what are the following changes you can expect in your implementation?
    a. Need to introduce major changes in the original application code
    b. Need to introduce new pointcuts
    c. Need to define new data structures to keep track of conversation

d. Lines of Code (LoC) and complexity of sample application may increase
e. Tangling and Scattering of sample application may increase
f. Require only minor change in implementation
g. Only need to modify some rules i.e., state machines etc., to accommodate new conversations
h. May expect some new bugs in the program
i. Overall debugging time would dramatically increase
j. Can reuse existing code to implement new changes

11. From scale 1-5, how would you rank the overall application after changes you implemented in Phase1 for code tangling (1 means fully tangled and 5 means two are totally independent)?

12. From scale 1-5, how would you rank the overall application after changes you implemented in Phase 1 for code scattering (1 means fully scattered in all classes and 5 means no scattering)?

13. How many hours did you spend to implement each of the following crosscutting concern?

APPENDIX E

EXTENDED APPLICATION DESCRIPTIONS

REQUIREMENTS FOR PHASE II

E.1. Connectionless Levenshtein Edit-Distance Calculator

This system allows user to enter two words into client console, which then requests a server to compute the Levenshtein Distance, LD, between the two words, wherein LD is the minimum number of single-character edits (insertion, deletion, and substitution) required to change one word into the other. For example, the LD between "kitten" and "sitting" is 3, since the following three edits change one into the other, and there is no way to do it with fewer than three edits:

- **k**itten → **s**itten (substitution of "s" for "k")
- sitt**e**n → sitt**i**n (substitution of "i" for "e")
- sittin → sittin**g** (insertion of "g" at the end)

This version of the design is similar to that in the initial application description. Figure E-1 describes the architecture, whereas Figure E-2 describes the interactions between Client and Edit-Distance Calculator. However, this version has the following differences from its initial draft:

- Communication between Client and Edit-Distance Calculator occurs using connectionless protocol or user datagram protocol (UDP).
- The message class uses the MessageID attribute of type UUID instead of RequestID and ResponseID.

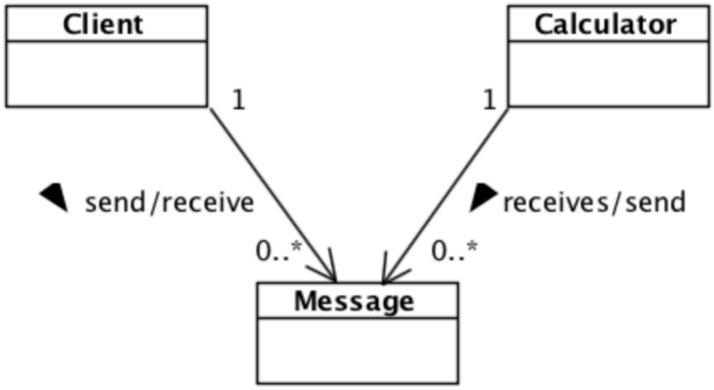

Figure E-1: Architecture Diagram of Levenshtein Edit-Distance Calculator

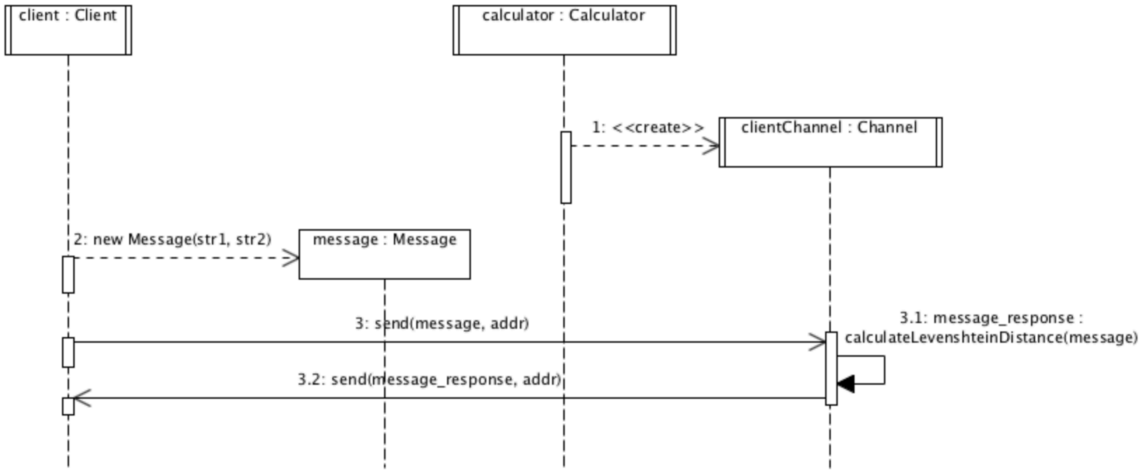

Figure E-2: Interaction Diagram between Client and Edit-Distance Calculator

E.2. File Transfer Protocol

FTP Client requests FTP Server for a list of available files and then sends a file download request to the server. The server sends the requested file in small chunks to the client.

This overall functionality is similar to that of initial application description. Figure E-3 shows an overview of the current architecture for this system whereas the UML Sequence Diagram in Figure E-4 shows this client-server interaction in more details. However, this version has following changes:

- *FileTransferAck* message is removed. Hence, client will not inform the server about the successful transfer of a file. After sending the last chunk of data, the Server terminates itself.

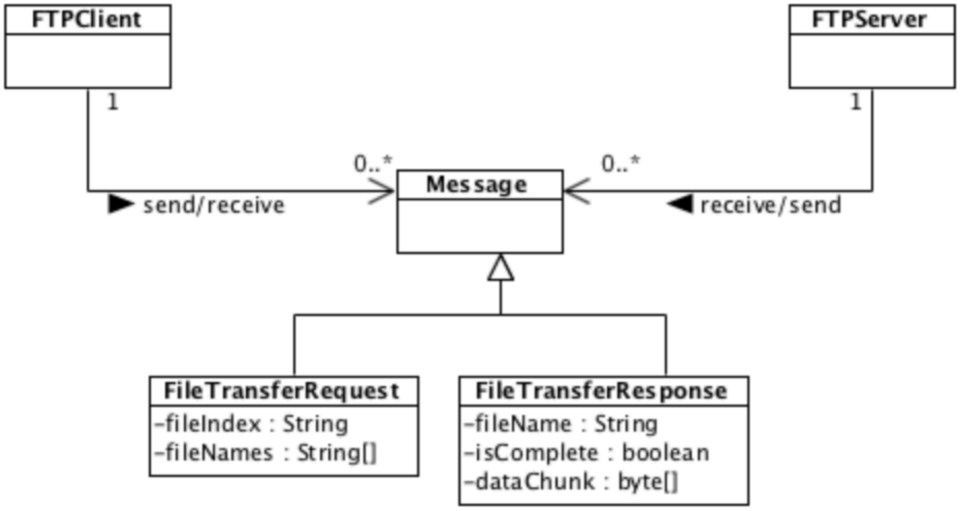

Figure E-3: Architecture Diagram for FTP

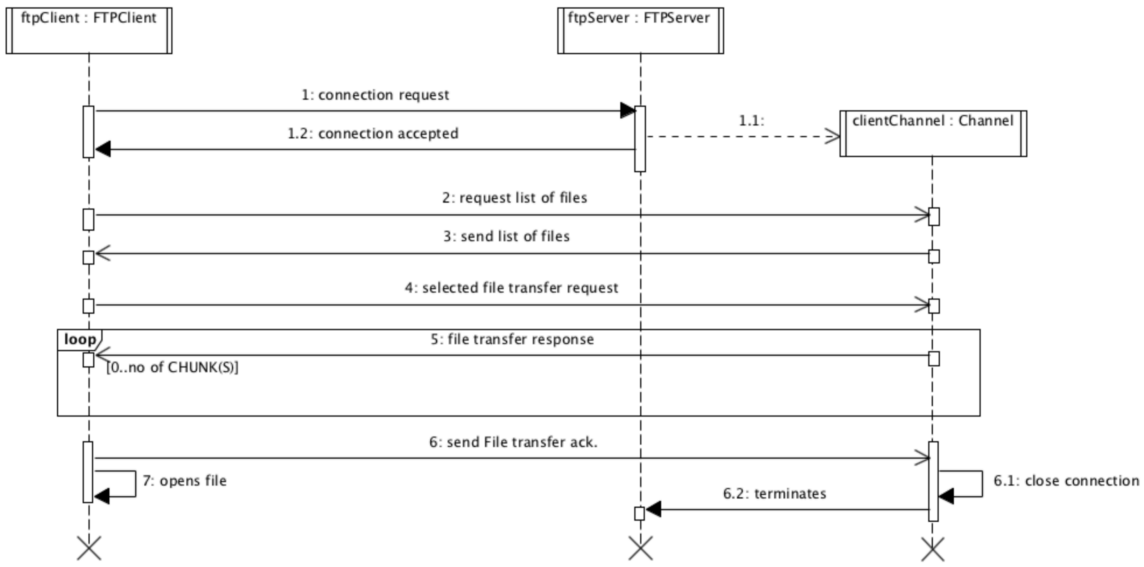

Figure E-4: Interaction Diagram between *FTPClient* and *FTPServer*

E.3. Weather Station Simulator

This example simulates a typical weather station consisting of three main components, i.e., *WeatherStationSensor, Transmitter* and a *Receiver.*

WeatherStationSensor, runs in a thread, generates weather-data readings at random intervals and temporarily stores them in a queue, accessible to the *Transmitter.* On receiving a request weather-data from the *Receiver* in random intervals, the *Transmitter* sends all of the data available in the queue, one weather-data reading at a time and in order, to *Receiver. Receiver* periodically sends more requests for weather data if it don't receive any data for some time period.

Receiver can requests the *Transmitter* to either SEND, PAUSE or STOP WeatherDataVector(s) as shown in Figure E-5.

- If *Receiver* sends a WeatherDataRequest of type SEND to each *Transmitter, Transmitter* receives the request, and starts sending the stored WeatheDataVector(s), one at a time. After transferring all the

WeatherDataVector(s), *Transmitter* sleeps unless *Receiver* notifies it again. When *Receiver* receives WeatherDataVector, it saves to a file and returns to the listening state. *Receiver* resends WeatheDataRequest of any value after random time interval.

- If *Receiver* sends PAUSE request, *Transmitter* interrupts sending of WeatherDataVector(s) and sleeps.
- If *Receiver* sends STOP request, *Transmitter* terminates itself.

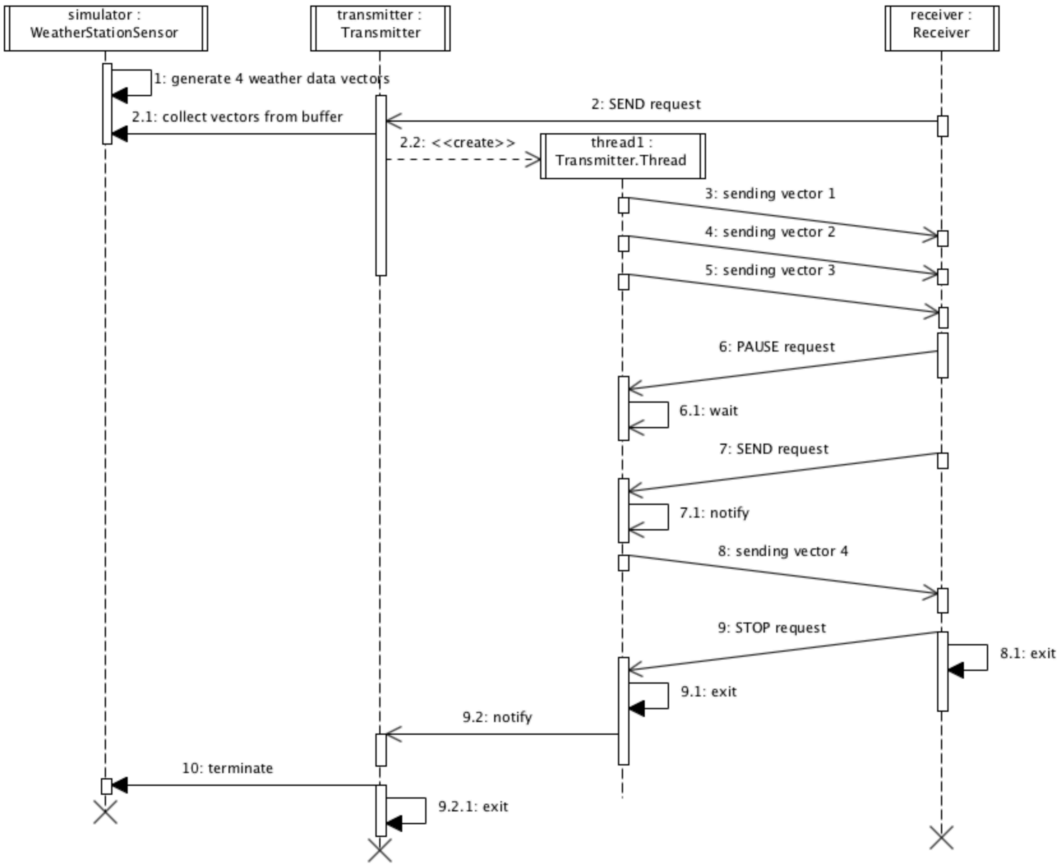

Figure E-5: Interaction Diagram between *Transmitter* (Two Threads) and Two *Receivers*

APPENDIX F

EXTENDED EXTENSIONS FOR PHASE II

F.1. Enhancements in the Performance Measure

Calculate the similar performance measurement statistics (Appendix B.2.) for the following programs as follows:

- *Levenshtein Edit-Distance Calculator:* A conversation is when a calculator receives a request and sends a response to the client
- *File Transfer Protocol:* A conversation is when a server receives the selected file transfer request and sends the last chunk of data to the server
- *Weather Station Simulator:* A conversation is when a *Receiver* sends a request to get weather related data readings and receives the first response from the *Transmitter* (see Enhancements for modification)

F.2. Enhancements in the Version Control

The version control is calculated using Message class attributes Sender version and *Receiver* version attributes, respectively.

F.3. Enhancements in the Encryption

Communication between KeyClient and KeyManager are implemented using UDPChannels.

APPENDIX G

QUESTIONNAIRE FOR PHASE II IMPLEMENTATION

Volunteer # _____

1. The phase 2 changes have following results on phase 1 changes?
    a. No effect
    b. Applications did not run properly
    c. Applications throw exceptions

2. To integrate phase 2 changes into phase 1 changes, you need to make the following code modifications?
    a. No change in implementation was required
    b. Need major changes such as creating new classes
    c. Need moderate changes such as creating new methods and variables
    d. Need minor changes such as modifying few existing methods and variables
    e. Overall scattering or tangling increased due to phase 2 application changes
    f. None of the above

3. While implementing the phase 2 features for phase 1 applications, which of the following did you find the most difficult?
    a. Adding crosscutting concerns to the applications design
    b. Deciding how to share data between previously existing sample application code and new code
    c. Debugging the applications with crosscutting concerns

d. Working with the *Java* implementation language or the IDE

e. Managing the complexity of the application

4. While implementing the phase 2 application changes, which of the following did you find the most difficult?

    a. Adding crosscutting concerns to the applications design

    b. Deciding how to share data between previously existing sample application code and new code

    c. Debugging the applications with crosscutting concerns

    d. Working with the *Java* implementation language or the IDE

    e. Managing the complexity of the application

5. What of the following was the most time consuming during implementation of phase 2 feature changes?

    a. Understanding the original applications and analyze the new requirements

    b. Designing the solutions

    c. Implementing the solutions

    d. Debugging the solutions

    e. Learning the tools (e.g., *Java,* an IDE)

    f. Learning AOP (not applicable for group 1)

    g. Learning *CommJ* (not applicable groups 1 and 2)

6. What of the following was the most time consuming during implementation of phase 2 application changes?
    a. Understanding the original applications and analyze the new requirements
    b. Designing the solutions
    c. Implementing the solutions
    d. Debugging the solutions
    e. Learning the tools (e.g., *Java,* an IDE)
    f. Learning AOP
    g. Learning *CommJ* (not applicable groups A)

7. While implementing your phase 2 changes in both applications and features, did your come across any of the following situations? (Select all that apply)
    a. Your changes introduced new bugs
    b. Your changes introduced new dependency among existing application components
    c. Tangling and scattering increased
    d. None of the above

8. If you were asked to refactor the phase 2 changes so it could be reused by other applications, which of following would you do?
    a. Redesign the application's structure, making major changes in the classes, their relationships, and responsibilities

b. Refactor the code to make minor improvements to the classes, their relationships, or responsibilities
c. Improve the implementation of individual methods, independent of changing the structure of the application, to improve readability or maintainability
d. Nothing – the implementation is ready for reuse

9. In phase 2, your original application (such as Edit-Distance Calculator and FTP) was implemented using connectionless communications. To implement this modification you made?
    a. Major changes
    b. Minor changes
    c. No different

10. In phase 2, your original application of WeatherStationSimulator was implemented using multiple *Receiver* s. To implement this modification you made?
    a. Major changes
    b. Minor changes
    c. No different

11. In phase 2, your original application (such as Edit-Distance Calculator and FTP) was implemented using JDK Sockets rather than JDK Channels. To implement modification you made?

a. Major changes
b. Minor changes
c. No different

12. Would your application be able to run standalone again if you remove the phase 2 feature changes from sample application code?
    a. Yes
    b. No
    c. Not sure

13. Would your application be able to run standalone again if you remove the phase 2 application changes from sample application code?
    a. Yes
    b. No
    c. Not sure

14. In order to implement the change in requirements for “Performance Measurement” feature such that a conversation is not only request-reply sequence but also a request-reply-acknowledgement (multi-step conversation), what are the following changes you made in your implementation?
    a. Need to introduce major changes in the original application code
    b. Need to introduce new pointcuts
    c. Need to define new data structures to keep track of conversation

d. Lines of Code (LoC) and complexity of sample application may increase
e. Tangling and Scattering of sample application may increase
f. Require only minor change in implementation
g. Only need to modify some rules i.e., state machines etc., to accommodate new conversations
h. May expect some new bugs in the program
i. Overall debugging time would dramatically increase
j. Can reuse existing code to implement new changes

15. From scale 1-5, how would you rank the overall application after changes you implemented in Phase2 for code tangling (1 means fully tangled and 5 means two are totally independent)?

16. From scale 1-5, how would you rank the overall application after changes you implemented in Phase 2 for code scattering (1 means fully scattered in all classes and 5 means no scattering)?

17. How many hours did you spend to implement phase 2 extension changes?

18. How many hours did you spend to implement phase 2 application changes?

APPENDIX H

DATA ASSESSMENT FROM THE SURVEYS

Based on our skill assessment survey in Appendix C, we gathered the following data about the background of participants in the experiment. The observations we make from the data support to our initial requirements about the selection and background of the experiment mentioned in Chapter 8.

H.1. Language Preferences of the Participants

Figure H-1 shows that all the participants selected only C# or *Java* as their preferred programming languages. Out of seven, four participants showed interest in *Java* and three in C#.

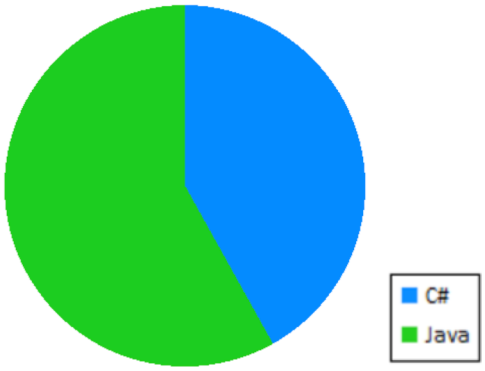

Figure H-1: Language Preferences of the Selected Participants

H.2. Programming Experience

H.2.1. Previous Programming Experience

All the participants had some previous programming experience. From the graph in Figure H-2, we can see that four Participants had 1-3 years of experience, two had over 5 years of experience and one had less than a year of experience.

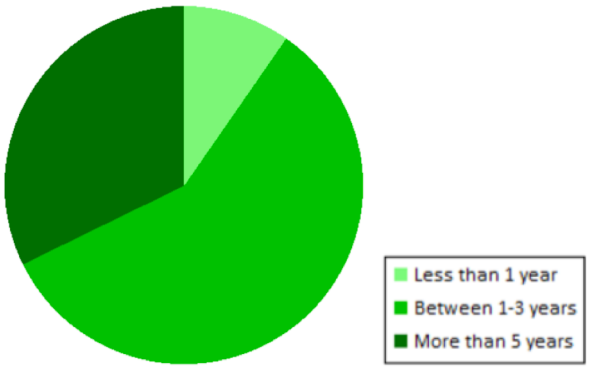

Figure H-2: Programming Experience of the Selected Participants

H.2.2. Quality of Experience

The graph in Figure H-3 shows us that the majority of the participants (four participants) had experience in developing programs with up to 1,000 – 10,000 LoC. Two participants had developed programs of over 10,000 LoC, and only one had developed less than 1,000 LoC.

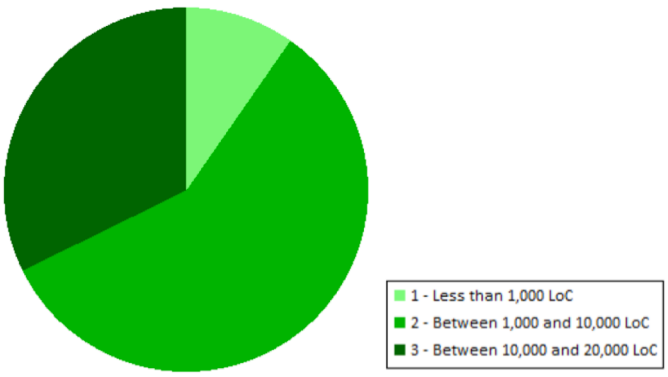

Figure H-3: Previous Projects LoC of the Selected Participants

H.3. Java/Software Engineering-Specific Skill Set

Figure H-4 illustrates for us the following observations about the participants.

- Almost 80% of the participantd had intermediate-level expertise in understanding and applying good design principles.
- Almost 80% of the participants had basic or no familiarity with network programming in *Java.* Hence, we arranged tutorials on network programming, and in later surveys, participants described themselves as having a sufficient grasp to implement the network programming-related tasks in the experiment.
- Almost 80% of the participants had only a basic familiarity with the multithreading concepts. Our tutorial on multi-threaded and network programming proved helpful for the participants to comfortably implement the required programming tasks in the experiment.
- Collectively, 90% of the participants were found to have intermediate or high expertise in understanding and applying UML.

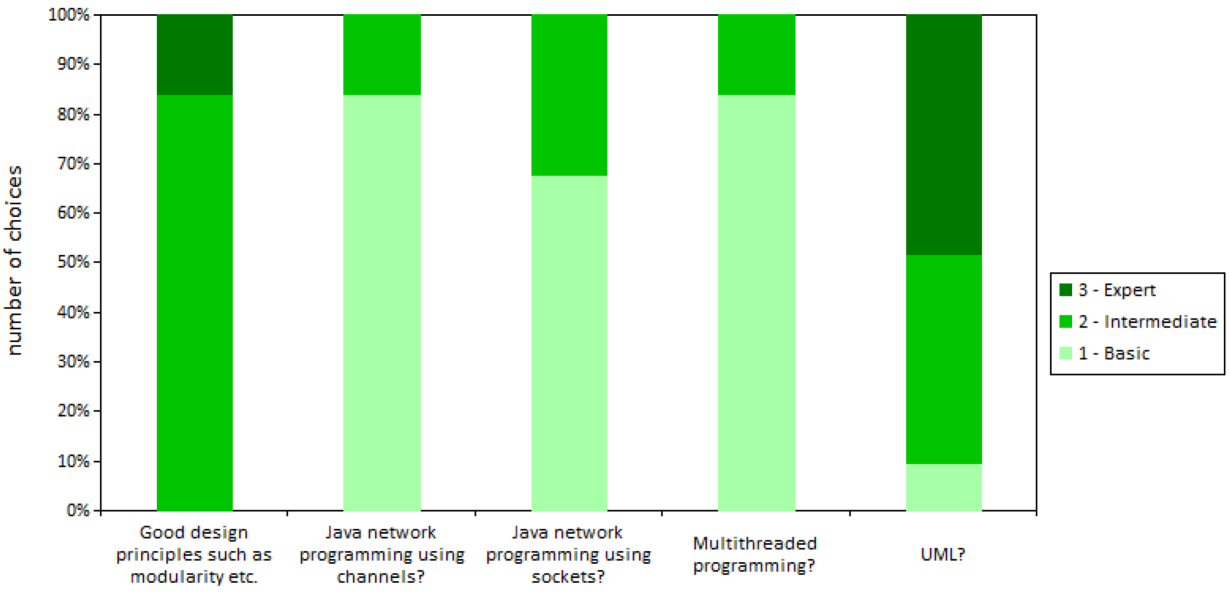

Figure H-4: Specific Skills Set of the Selected Participants

From the data in the above graphs, we can easily conclude that participants shared a common background in object-oriented concepts, previous programming experience, and level of projects completed in the past, as well as understanding and applying good software engineering principles. Hence, the selected participants were found to sufficiently fulfill the requirements related to the selection of the developers in Section 8.3.1.

APPENDIX I

DOCUMENTS FOR THE

RESEARCH EXPERIMENT APPROVAL

I.1. CITI Passing Report

As per requirements of IRB, the student researcher was supposed to pass the Human Research Experiment Training course (See Figure I-1 below).

COLLABORATIVE INSTITUTIONAL TRAINING INITIATIVE (CITI)
HUMAN RESEARCH CURRICULUM COMPLETION REPORT
Printed on 08/12/2013

LEARNER	Ali Raza (ID: 3645300) 09 Aggie Village Apt H Logan UTAH USA
DEPARTMENT	Computer Science
PHONE	435-225-3723
EMAIL	ali.raza@aggiemail.usu.edu
INSTITUTION	Utah State University
EXPIRATION DATE	08/11/2016

SOCIAL & BEHAVIORAL RESEARCH MODULES

COURSE/STAGE:	Basic Course/1
PASSED ON:	08/12/2013
REFERENCE ID:	10919905

REQUIRED MODULES	DATE COMPLETED	SCORE
Introduction	08/10/2013	No Quiz
Conflicts of Interest in Research Involving Human Subjects	08/11/2013	4/5 (80%)
History and Ethical Principles - SBE	08/11/2013	4/5 (80%)
Defining Research with Human Subjects - SBE	08/11/2013	4/5 (80%)
The Regulations - SBE	08/12/2013	5/5 (100%)
Assessing Risk - SBE	08/12/2013	5/5 (100%)
Informed Consent - SBE	08/12/2013	4/5 (80%)
Privacy and Confidentiality - SBE	08/12/2013	5/5 (100%)
Utah State University	08/12/2013	No Quiz

For this Completion Report to be valid, the learner listed above must be affiliated with a CITI Program participating institution or be a paid Independent Learner. Falsified information and unauthorized use of the CITI Progam course site is unethical, and may be considered research misconduct by your institution.

Paul Braunschweiger Ph.D.
Professor, University of Miami
Director Office of Research Education
CITI Program Course Coordinator

Figure I-1: CITI Passing Report

I.2. Research Experiment Invitation Letter

Following invitation letter was sent to the interested participants in order to get their voluntarily approval to participate in our research experiment.

LETTER OF INVITATION TO PARTICIPATE IN *A* RESEARCH STUDY

Investigation into the Benefits of weaving aspects into Inter-process Communications (IPC)

Dated: 11/07/2013

Dear Students,

We are in process of conducting a research experiment to measure the reusability and maintainability for an aspect-oriented framework, called *CommJ,* with respect to *AspectJ*.

We believe you a good candidate for our research study because you meet the following criteria:

- You are enrolled for a degree program in Computer Science
- You have good exposure of OOD and Unified Modeling Language (UML)
- You have taken at least one programming course in *Java*
- You have taken at least one software-engineering class
- You have exposure to multi-threaded concepts in *Java*

By helping us in our research study, you are contributing in the advancement of software engineering tools and methods for network applications. In addition to

receiving a $200 stipend, you may also receive the following benefits by participating in the study:

- New skills in aspect-oriented programming
- An opportunity to learn a new software development framework, namely *CommJ*
- Additional practice and experience with object-oriented design and software engineering principles

Completing your part of the study will involve the task listed below and should take around 30 hours of your time:

- Enhance three existing applications (written in *Java)* to meet the requirements for three new extensions
- Update the three applications to meet a second set of requirements
- Record your observations in a journal throughout the development
- Completing questionnaires before and after each implementation phase

We look forward to you participation. If you have any questions about the experiment or your role, please contact Dr. Stephen Clyde (PI) at (435) 797-2307/Stephen.Clyde@usu.edu and Ali Raza (student researcher) at (435) 225-3723/ali.raza@aggiemail.usu.edu.

Regards,

Dr. Stephen Clyde (Principal Investigator)

Ali Raza (Student Researcher)

I.3. Experiment Approval Letter from Institutional Review Board (IRB)

IRB evaluated and approved the research experiment application. Approval letter is shown in Figure I-2 below.

Institutional Review Board

USU Assurance: FWA#00003308

UtahState University
OFFICE OF RESEARCH AND GRADUATE STUDIES

Full Accreditation

Exemption #2

Certificate of Exemption

FROM:

Melanie Domenech Rodriguez, IRB Chair

True M. Rubal, IRB Administrator

To: *Stephen Clyde, Ali Raza*
Date: *October 10, 2013*
Protocol #: *5387*
Title: *Investigation Into The Benefits Of Weaving Aspects Into Inter-Process Communications*

The Institutional Review Board has determined that the above-referenced study is exempt from review under federal guidelines 45 CFR Part 46.101(b) category #2:

> *Research involving the use of educational tests (cognitive, diagnostic, aptitude, achievement), survey procedures, interview procedures or observation of public behavior, unless: (a) information obtained is recorded in such a manner that human subjects can be identified, directly or through the identifiers linked to the subjects: and (b) any disclosure of human subjects' responses outside the research could reasonably place the subjects at risk of criminal or civil liability or be damaging to the subjects' financial standing, employability, or reputation.*

This exemption is valid for three years from the date of this correspondence, after which the study will be closed. If the research will extend beyond three years, it is your responsibility as the Principal Investigator to notify the IRB before the study's expiration date and submit a new application to continue the research. Research activities that continue beyond the expiration date without new certification of exempt status will be in violation of those federal guidelines which permit the exempt status.

As part of the IRB's quality assurance procedures, this research may be randomly selected for continuing review during the three year period of exemption. If so, you will receive a request for completion of a Protocol Status Report during the month of the anniversary date of this certification.

In all cases, it is your responsibility to notify the IRB prior to making any changes to the study by submitting an Amendment/Modification request. This will document whether or not the study still meets the requirements for exempt status under federal regulations.

Upon receipt of this memo, you may begin your research. If you have questions, please call the IRB office at (435) 797-1821 or email to irb@usu.edu.

The IRB wishes you success with your research.

4460 Old Main Hill *Logan, UT 84322-4460* *PH: (435) 797-1821* *Fax: (435) 797-3769* *WEB: irb.usu.edu* *EMAIL: irb@usu.edu*

Figure I-2: IRB Approval Letter