CommJ: An Extension to AspectJ for Improving the Reuse and Maintainability of Communication-related Crosscutting Concerns

Ali Raza
SaaS Cloud Engineer
ItsOn Inc.
Silicon Valley, California, USA
ali.raza@itsoninc.com

Stephen Clyde
Computer Science Department
Utah State University
North Logan, Utah, USA
Stephen.Clyde@usu.edu

***Abstract*—This paper presents advances of research on CommJ, a framework for weaving communication-aware aspects into application code. Specifically, it presents a simplified *Universe Model for Communication* (UMC) and an enhanced implementation of CommJ. It also summarizes a preliminary experience that tests seven hypotheses about how CommJ might improve reusability and maintainability of software applications that rely on network communications. The summary includes a description of a quality model consisting of factors that impact reusability and maintainability, attributes that the factors depend on, and metrics for assessing those attributes. The experiment was a two-group study involving seven aspect-oriented programmers. Despite the small number of study participants, the experiment yielded encouraging results about CommJ's potential. Specifically, CommJ can improve reusability and maintainability of application code when there are communications-related crossing-cutting concerns.**

Keywords – aspect orientation; aspect-oriented programming languages; AspectJ; communications; cross-cutting concerns; software reuse; software maintainability.

I. Introduction

Inter-process communications are ubiquitous in today's software systems, yet they are rarely treated as first-class programming concepts. Consequently, developers have to implement communication protocols manually using primitive operations, such as connect, send, receive, and close. For many 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 between clients and servers. With Passive FTP, a 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., start an instance of the Passive FTP protocol, by sending a connect request to the server on that port. The server sets up a dedicated port, 2024 in this example, and sends its number back to the client. The client connects to that port and the server sends back an acknowledgment. At this point, two processes can start exchanging data on this dedicated communication channel. The arrows in Figure 1 illustrate this initial sequence of messages.

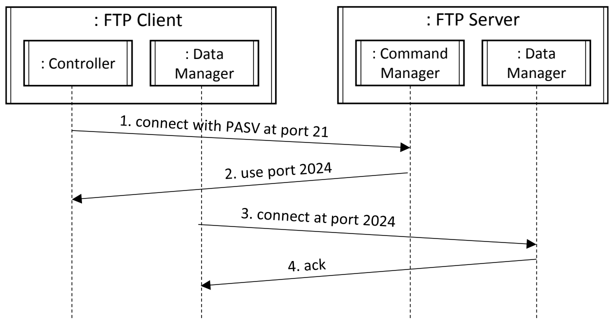


Figure 1. The Starting of a Passive FTP Conversation

Neither the client's nor the server's side of the conversation is trivial. In fact, both usually execute different parts of the conversation on different threads. For example, Figure 1 shows two threads for a FTP server and two threads for a FTP client. Although multi-threading has many advantages, it can create complexities while trying to follow a conversion's execution in the code because different parts of the conversation end up being handled by different components in the code.

A distributed system with concurrent conversations based on one or more non-trivial communication protocols may be further complicated by other communication-related requirements, such as logging, detecting network or system failures, monitoring congestion, balancing load across redundant servers, and supporting multiple versions of one or more of the protocols [1][2].

From a communication perspective, these concerns are examples of what Aspect Orientation (AO) refers to as crosscutting concerns, because they pertain to or cut through multiple parts of a core or base system. Implementing one or more of these concerns without careful attention to encapsulation, modularization, cohesion, and coupling can cause undesirable scattering and tangling of code.

AO, which first appears in the literature in 1997 [3][4], reduces scattering and tangling of code by encapsulating crosscutting concerns in first-class programming constructions, called aspects [5]. An aspect is an Abstract Data Type (ADT), just like classes in strongly-typed, class-based, object-oriented programming 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, which are temporal points during the execution of the system when weaving of advice may take place. Each joinpoint corresponds to static place in the source code, called a shadow [5].

AspectJ is an extension to Java for aspects and, like many other AOPLs and Aspect-oriented Frameworks (AOFs) [5][6][7][8], it allows programmers to weave advice for crosscutting concerns into joinpoints that correspond to various programming construct, such as constructor calls/executions, methods calls/executions, class attribute references, and exceptions. For documentation on AspectJ, see *The AspectJ Project* website [5].

Since aspects are just special ADTs, it is possible for skilled software developers using traditional object orientation (OO) to implement classes that do basically the same thing. However, these classes would have to manage all the joinpoint contexts and weaving of crosscutting behaviors explicitly. Furthermore, hooks into crosscutting behaviors would most likely have to be introduced into the core application code. In other words, the real difference between AO and OO is that AO offers a convenient mechanism for separating crosscutting concerns from core functionality. It encourages obliviousness, which is the idea that core functionality should not have to know about crosscutting concerns [9][10]. With obliviousness, programmers should be able to add or remove the crosscutting concerns at build time without changing any source code.

The problem is that AspectJ and other AOPLs do not support the weaving of advice into core high-level application abstractions, such as conversations among processes in a distributed system, since those abstractions are based on run-time context information beyond code constructs, a single thread flow of execution, or its call stack. This paper introduces an extension to AspectJ, called CommJ [1][2] that allows developers to weave crosscutting concern into conversations in a modular and reusable way, while keeping the core functionality oblivious to those concerns.

We elaborate on the problem and review related literature in Section II. Then, in Section III, we formalize the notion of communication joinpoints and introduce CommJ. Next, Section IV demonstrates the feasibility and utility of CommJ by describing a library of reusable communication aspects and providing examples from a non-trivial sample application.

To explore CommJ's utility as a valuable extension to AspectJ, we conducted a preliminary experience that measured the quality of applications and extensions to applications built with CommJ compared to the same built with only AspectJ. Section V describes the quality model that we used for the comparison and evaluation of the application software. It is an adaption of the Comparison Quality Model by C. Sant'Anna et al. [11]. Sections VI & VII explain the hypotheses and experimental method, while Section VIII presents the results of the experiment along with our interpretations and conclusions for each hypothesis.

Overall, the experiment provides preliminary evidence that the applications written with CommJ are more cohesive and oblivious and that they have less scattering and tangling of cross-cutting concerns then their AspectJ-only comparison applications. Furthermore, those using CommJ were more loosely coupled, less complex, and smaller in size. The results are encouraging and provide ample motivation to continue work on CommJ and to pursue other opportunities for weaving aspects into application-level abstractions. Section IX summarizes the contributions of this paper and discusses future work.

II. Background and Related Work

To explain CommJ and its contributions, it is first necessary to establish a foundation of background concepts and related work in four areas: The AOP paradigm, AOP development tools (i.e, languages and frameworks), communications, and cross-cutting concerts with respect to communications.

A. *The AOP Paradigm*

In general, a skilled programmer can do anything in an OO programming language (OOPL) that could be done in an AOPL by making careful design decisions that encapsulate crosscutting concerns in well-modularized classes and hooking those features into the base application. To do this, a programmer could use software constructs, such as delegates, callbacks, and events, or apply various design patterns, like the Strategy, the Decorator, and the Template Method patterns [13]. However, the developer may still end up with code tangling and scattering, unnecessary coupling, lack of obliviousness, and compromised flexibility. AO provides an elegant way of weaving new behaviors into existing code, such that their functionality is less scattered, tangled, and decoupled from the base application, without compromising that functionality.

With AO, programmers should only need modular reasoning to discover the code and structure of the crosscutting concerns; whereas they would most likely need global reasoning when using traditional OO techniques [13]. Additionally, when using only OO techniques, separating out tangled code from core functionality can cause inheritance anomalies [14]. AO programmers, on the other hand, can refactor tangled code by moving it into loosely coupled aspects. So, the attraction of AO is not that a developer can do more, but that a developer can do certain things better, particularly in terms of modularizations or crosscutting concerns with less scattering and less tangling.

B. *AOP Languages and Framework*

Other techniques that address the same problems solved by OO, including Monads [15], Subject-oriented Programming [16][17], Reflection [18][19], Mixins [20], and Composition Filters [18]. The AO approach seems to have risen to the top as the most influential because it allows for better modularity of crosscutting concerns and it does not alter or violate principles of the OO paradigm.

There are many AOPLs and AOFs available today, such as AspectJ [5], AspectWorkz [6], Spring AOP [8] and JBoss AOP [7]. Though they are semantically similar in terms of their aspect invocation, initialization, access and exception handling routines, they differ in programming constructs, syntax, binding, expressiveness, approaches to advise weaving, static or dynamic analysis, and their overall acceptance in academia and industry. Currently, AspectJ (powered by IBM) is the de facto standard and the most widely used AOPL. Perhaps, this is because it is an extension to Java and takes advantage of Java class libraries and development environments.

None of the these AOPLs and AOFs allow developers to consider a conversation as a context into which cross-cutting concerns may be woven. They all focus on either compile-time contexts, like methods and classes) or primitive run-time contexts, like the objects and call stacks. To allow conversations to be contexts, without forcing programmers to create the necessary infrastructure manually as part of the application code, an AOF for communications would have to define model of communications and then automatically track individual conversations.

C. *Communications, Conversations, and Protocols*

In general, inter-process communications are either connection-oriented or connectionless. Connection-oriented communications require two concurrent processes to establish a communication link before exchanging data. This style of communication is very much like a person-to-person telephone call. In contrast, with connectionless communications, one process can send a message to another process without knowing whether that process is ready to receive the message or if it even exists yet. This style of communication is like traditional postal mail. Communication subsystems or libraries, like the JDK's Channels and Sockets, typically support both styles of communication.

A conversation is a series of interactions between two or more processes for some purpose. It may include the formation of a connection (only for connection-oriented communications), exchange of messages among the processes, and the termination of the connection (again, only for connection-oriented communications). A conversation is like a phone call with a doctor's office to setup an appointment or a series for postal mailings that were necessary for the signing of a contract. Conversations can last for just a millisecond or go on for days.

Like a formal interaction between two parties signing a contract, an electronic conversation between processes follows a protocol that governs the expected behavior of the participating processes. Some protocols are symmetrical, meaning that all participants follow the same rules. However, it is more common for protocols to be asymmetrical, meaning that each participant acts according to the role it is playing. Many protocols, like the Passive FTP example mentioned earlier, involve two roles: a conversation initiator and responder. Sometimes, these roles are simply referred to as client and server. However, these terms have broader meanings that imply other software architectural issues beyond communications, so we avoid them here.

Implementations of communication details can vary depending on the underlying communication libraries, e.g., Channels and Sockets. These differences are, however, only of secondary importance and can be easily supported by different adapters in CommJ implementation. So, further explanation of diversity and subtleties of the various communication implementation techniques are beyond the scope of this paper.

D. *Crosscutting Concerns in Communications*

Despite AspectJ's rich set of pointcut designators, there is still a weakness relative to weaving crosscutting concerns into communications. Specifically, AspectJ and any other similar AOPLs or AOFs do not work with conversations directly. Specifically, they do not track individual conversation contexts or link together messages of the same conversation. Consequently, programmers cannot weave behaviors directly into individual conversations. Furthermore, since the execution of conversation may be spread across multiple software components and multiple threads, tracking individual conversations is beyond what language-construct-based aspect weaving can accomplish.

Consider a communication-related crosscutting concern that involves tracking the total time for all connectionless conversations in a distributed application. If a programmer wants to implement crosscutting concern in AspectJ, he or she would have to implement some advice for the conversation's initiation that would capture the time when the first message was sent, as well as other advice that would capture the time when the last message was received and then compare the two times. However, send and receive logic for the conversation may be in separate code modules, may be separated in the execution flow by an unpredictable amount of time, and may even be handled on separate execution threads. Furthermore, a process may start or participate in many conversations at the same time, and the advice would have to manually correlate the first message of particular conversation with the last message of that conversation. In a nutshell, the programmer would have to build all of the message tracking and correlation objects into the aspect and its advice.

III. COMMJ

To enable the weaving of advice into individual conversations, we first define a general model, i.e., the Universe Model of Communication (UMC), for connection-oriented and connectionless communications and use it as a basis for formalizing the notation of communication joinpoints. We then implement CommJ according to the

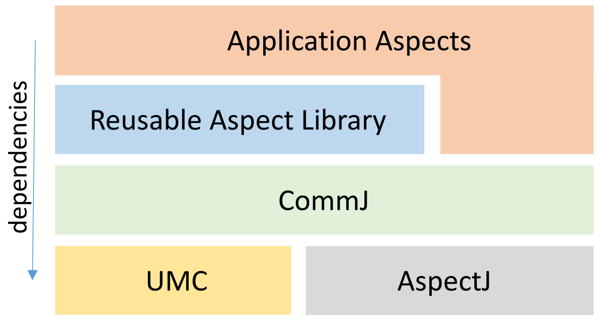


Figure 2. CommJ and its associated components

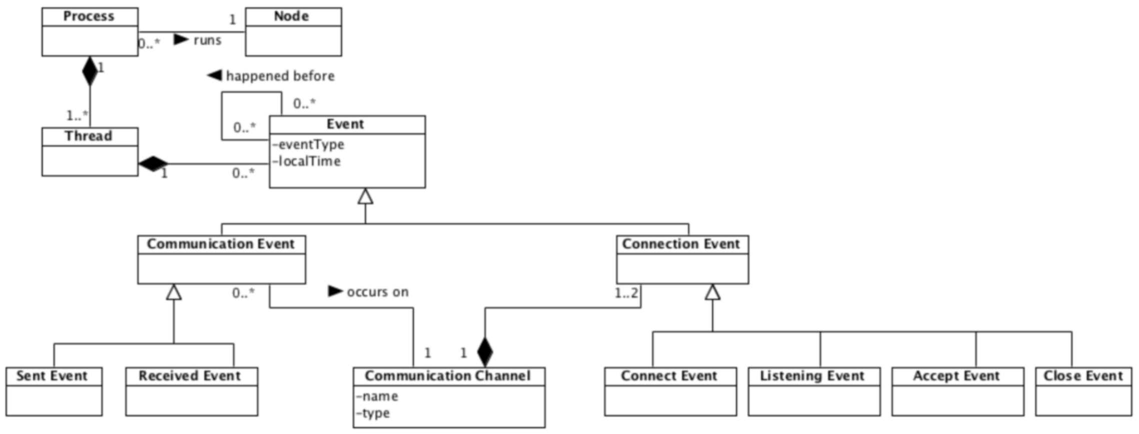


Figure 3. UMC for Events

UMC and on top of the aspect capabilities provided by AspectJ (see Figure 2). The CommJ implementation provides a) base aspects with abstract pointcuts for communications, independent of the underlying communications subsystem and b) behind-the-sense components to track individual conversation contexts at runtime. Application programmers simply include the CommJ library into their build and create their own communication aspects that inherit from the base CommJ aspects. To help them integrate common communication-related cross-cutting concerns into their application, we also provide a Reusable Aspect Library (RAL). The rest of this section describes the communications model, communication joinpoints, and the CommJ library in more detail. Then Section IV highlights some of the aspects in the RAL and shows how that can be used in a sample application.

A. *A Universe Model of Communications*

The UMC describes a minimal set of general concepts that cover both connection-oriented and connectionless communications provided by most communication systems. In doing so, it models events, threads, messages, conversations, and protocols, as well as the relationships among these concepts.

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 (not shown in Figure 3). A Communication Event is the happening of something (related to send or receive) in message-based communications, at a particular point in time. It is 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; an event can contain or be associated with many other events. For example, in a distributed application, a thread T1 can send a message which corresponds to a send event. Eventually, that can lead to a receive-message event on some other thread T2. The relation between these two events is modeled by the "happened before" relationship on Event in Figure 3.

Connection Events are happenings related to the setting up of communication channels, and are specialized into four types:

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

UMC does not need to include exception events explicitly because AspectJ already defines a rich set of pointcuts for defining crosscutting concerns that involve exceptions.

A Thread can instantiate and encapsulate 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, which in turn may have multiple threads.

2) *Conversations*

In general, a conversation from single process's perspective is a sequence of messages that follow communication rules that either comprise all or part of exchange with other process:

A. an entire conversion from a process's perspective (see the bracket sequence, A, in Figure 4)
B. any sequence of send or receive events in the conversation as seen by a process (see B in Figure 4)
C. a single send or receive event in a conversation (C in Figure 4)

In Figure 5, we see that each conversation in UMC can use a set of Communication Events. A Communication Event occurs on a Communication Channel and is indirectly

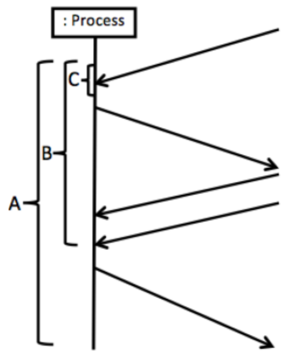


Figure 4. Conversations in *UMC*

associated with a protocol through it conversation to which the event belongs. A conversation is also capable of keeping track of Communication Events that occur in a multithreaded application with multiple channels.

With CommJ, a distributed application can consider a conversation all or just subset of the messages exchanged between specific processes, previously illustrated in Figure 4. This gives developer a freedom to organize the conversations in a manner that seems appropriate for the application and the freedom to use virtual any kind of communication protocol or pattern.

3) Channel

Every Conversation happens on a Channel (Figure 5). 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., JDK's Sockets and Channels, into higher level concepts that are consistent across platforms. In design pattern terms, the UMC's Communication Channel is like an Adapter [12] to underlying communication mechanisms, but for crosscutting concern.

4) Messages

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 be associated with at most one send and one receive event. Further, Messages and Communication Events follow similar specialization hierarchies; from a process's perspective both are specialized into send and receive types. An instance of Message received keeps track of its Received Event, and a Message sent knows about its Sent Event.

All CommJ applications derive their specific message classes from the base Message class (see Figure 6), which is in the CommJ Infrastructure. The Message class contains getter and setter methods for the properties shown in Figure 6. Collectively, the first five properties are referred to Message Identifying Information (MIF). These five elements provide the necessary information to identify the context of any message, thus enabling CommJ to create and manage conversation metadata, represented by the Conversation class in Figure 5.

The CommJ Infrastructure implements abstract Message with an interface, call IMessage. CommJ then dynamically introduces it into the core software during aspect initialize (see Section IV.A). The interface IMessage is the only direct dependency between the core application and CommJ.

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 responder while handling connection events. An initiator can handle only connect and close events, whereas a responder can handle listen, accept and close events, respectively. Figure 7 illustrates the connection-related concepts in UMC.

IV. COMMJ ASPECT LIBRARY AND SAMPLE APPLICATION

This section describes the general architecture of CommJ along with some fundamental concepts, mostly about low-level design and implementations. Finally, it discusses some sample applications, developed using CommJ.

A. Joinpoints

The UMC serves as a foundation for formalizing communication joinpoints, which fall into two general categories: communication joint points and connection-related join points, respectively.

1) Communication Joinpoints

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

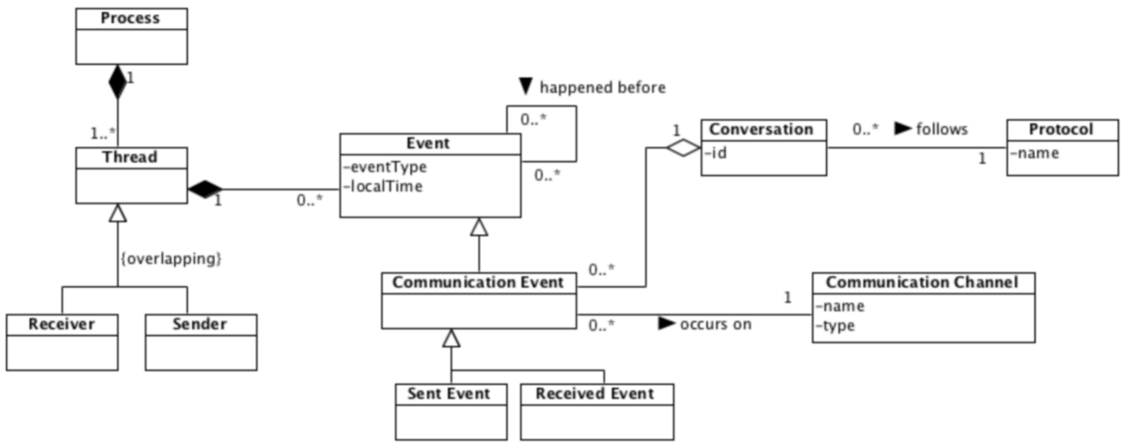


Figure 5. UMC for Conversations

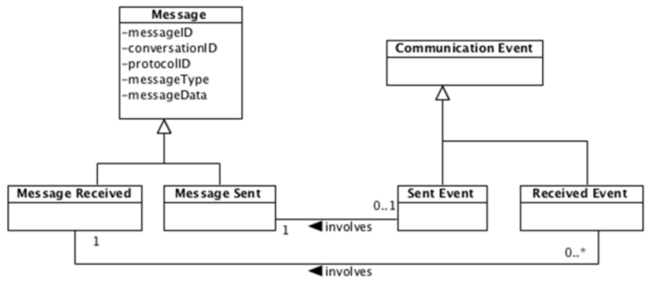


Figure 6. UMC for Messages

conversations to AspectJ as possible contexts. Unlike AspectJ contexts, however, a conversation is not tied to a single programming construct but to the runtime abstraction of an inter-process conversation.

Figure 8 represents different kinds of message related joinpoints in *CommJ*. A Send Event JP, 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, where as a Receive Event JP is related to receiving of data respectively. Request Reply Conversation JP, represents a joinpoint for complete conversations, but they follow basic request-reply protocols. It contains a Send Event JP and a Receive Event JP. A Send Event JP keeps track of message Id whereas a receive Event JP records a response Id 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 Send Event JP and adds it to the communication registry, which contains communication joint points. Similarly on receiving a message, it creates an instance of a Receive Event JP and finds a Send Event JP from the registry where the message Id of the former equals the response Id of the later. Finally, Multi-step Conversion JP, represents joinpoints for across multiple events or for entire conversations. Multiple send and receive events are modeled using a state machine in a Multi-step Conversation JP.

2) Connection Joinpoints

The other types of joinpoints are connection-related sequence of events such as connect, accept, listen, and close events. Connection joinpoints in CommJ are either owned by an initiator or a responder (see Figure 9 for more details about the following types of connection-related joinpoints in *CommJ*).

An initiator creates a Connect Event JP. It encapsulates the connection information related to underlying sockets and

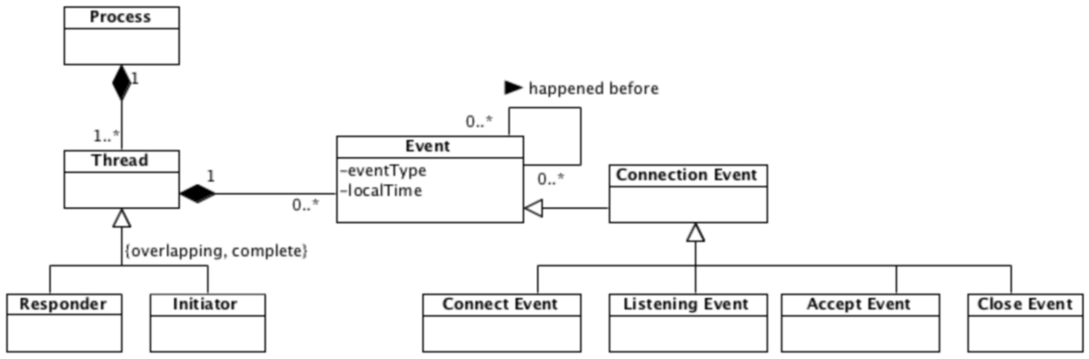


Figure 7. UMC for Connections

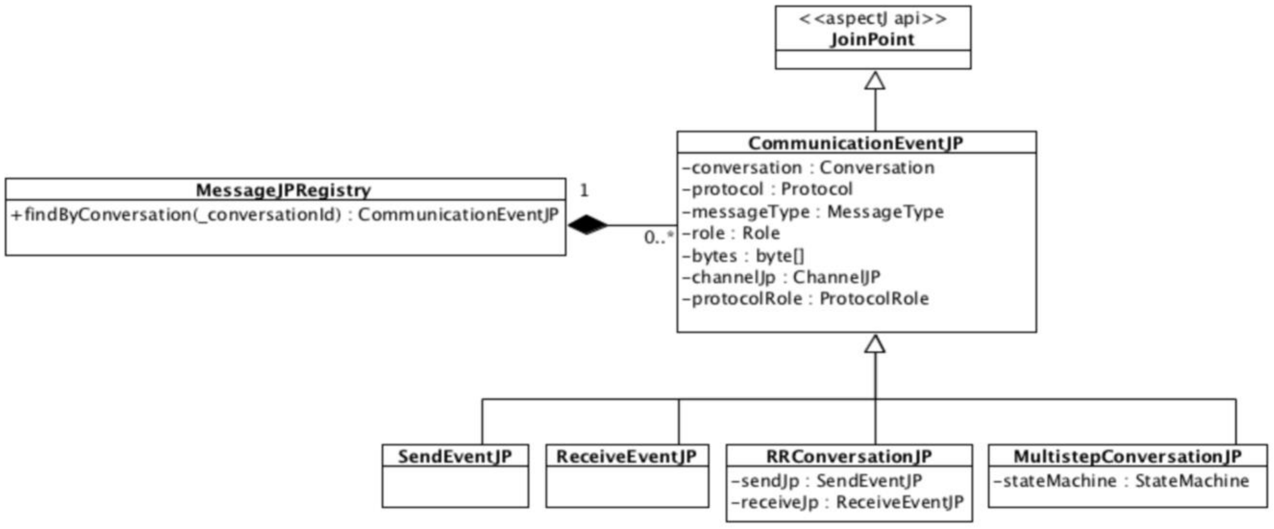
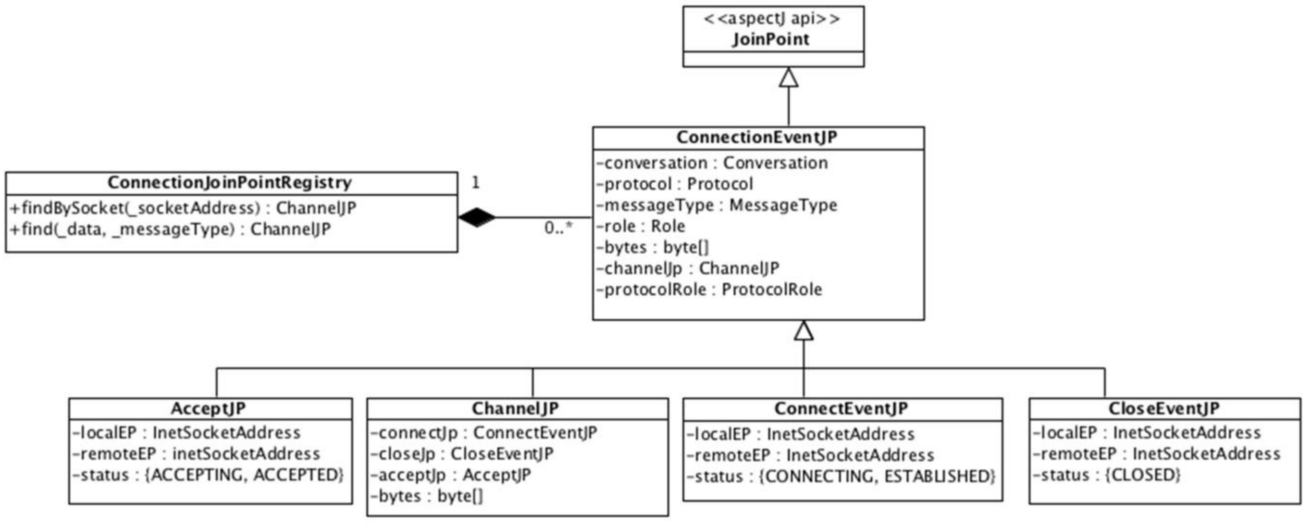


Figure 8. Communication Joinpoint and Registry



<<aspectJ api>>
JoinPoint
ConnectionEventJP
-conversation : Conversation
-protocol : Protocol
-messageType : MessageType
-role : Role
-bytes : byte[]
-channelJp : ChannelJP
-protocolRole : ProtocolRole
ConnectionJoinPointRegistry
+findBySocket(_socketAddress) : ChannelJP
+find(_data, _messageType) : ChannelJP
1
0..*
AcceptJP
-localEP : InetSocketAddress
-remoteEP : inetSocketAddress
-status : {ACCEPTING, ACCEPTED}
ChannelJP
-connectJp : ConnectEventJP
-closeJp : CloseEventJP
-acceptJp : AcceptJP
-bytes : byte[]
ConnectEventJP
-localEP : InetSocketAddress
-remoteEP : InetSocketAddress
-status : {CONNECTING, ESTABLISHED}
CloseEventJP
-localEP : InetSocketAddress
-remoteEP : InetSocketAddress
-status : {CLOSED}

Figure 9. Connection Joinpoint and Registry

channels along with their local and remote addresses. Responder creates an Accept JP on receiving a connection request from the initiator. Both the initiator and responder instantiate a Close Event JP when a connection closes.

Channel JP acts like a bridge between communication joinpoints and connection joinpoints. It also maintains links between a responder Accept JP and an initiator Connect Event JP. Additionally, a Channel JP Registry is used to correlate different connection-related events that belong to the same conversation.

B. *Joinpoint Trackers*

Behind the scenes, CommJ relies on JoinpointTrackers, which are monitors [13] 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, Joinpoint Trackers 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: message-joinpoint trackers and connection-joinpoint tracker.

1) Message Joinpoint Trackers

The Message Joinpoint Tracker (see Figure 10) crosscuts the send and receive events for both reliable and unreliable communications in the core application and defines a set of pointcuts in the simple send and receive abstractions. Message Joinpoint Tracker is an aspect that hides communication related abstractions in the core application.

The Message Joinpoint Tracker aspect defines pointcuts in the send and receive abstractions (Figure 11) by overcoming the syntactic and semantic variations, defined in JDK 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, Message Joinpoint Tracker creates two clean, well-encapsulated communications related

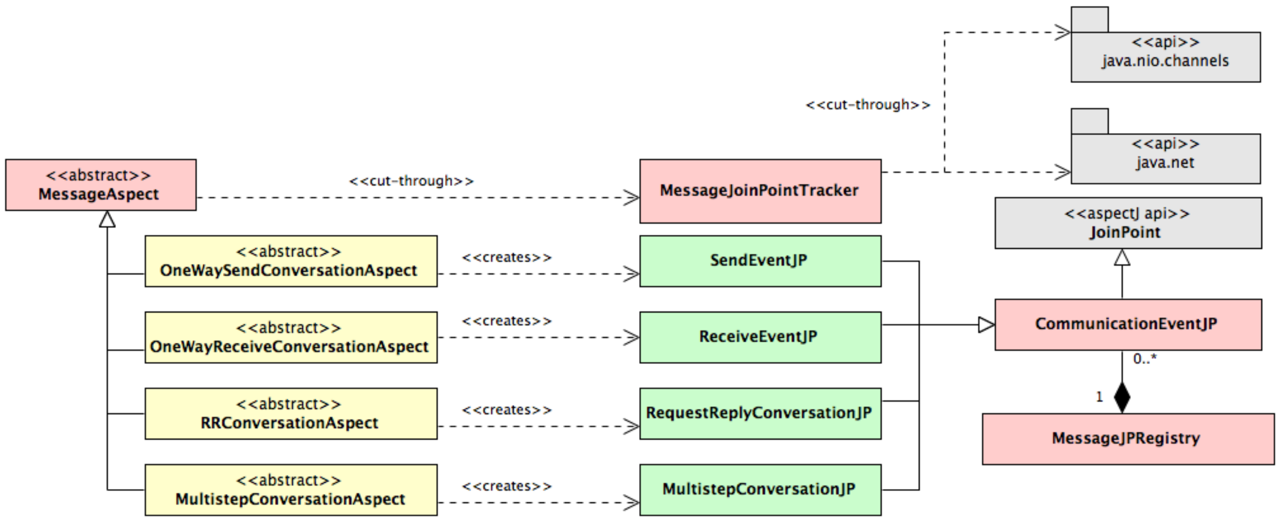


Figure 10. *CommJ* Message Event Join Points and Reusable Aspects

```
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) ||
    call(* DatagramChannel+.read(ByteBuffer)) && target(_channel) && args(_buffer);
                                                          ....
                                                         }
```

Figure 11. *CommJ* Message Event Join Points and Aspects

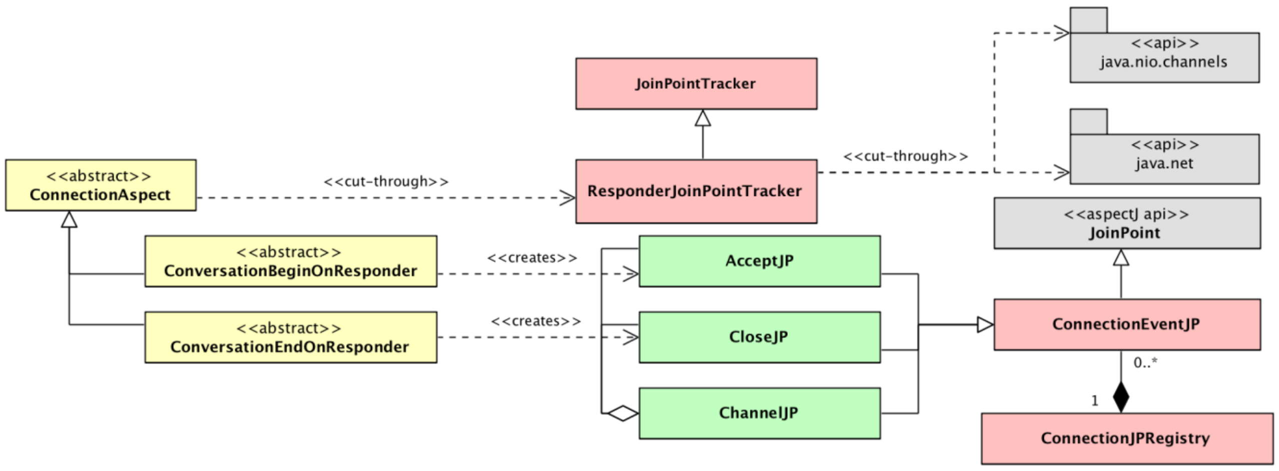


Figure 12. Responder Joinpoint and Base Aspects

abstractions for all types of read and write operations.

2) *Connection Joinpoint Trackers*

The Message Joinpoint trackers are categorized into Initiator Joinpoint Tracker and Responder Joinpoint Tracker, which 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.

The Responder Joinpoint Tracker, defines two simple pointcuts, i.e., Accept and Close, where Initiator Joinpoint Tracker, defines three pointcuts, i.e., Channel Connect, Channel Connect Finish and Channel Close pointcuts. These two trackers manage all connection-related abstractions and styles related to both the responder and initiator for connectionless and connection-oriented communications. Figures 12 & 14 describe the general architecture of responder and initiator, and Figures 13 & 15 present their code snippets.

```
public aspect ResponderJoinPointTracker {
   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);


  public pointcut ChannelOpen(DatagramChannel _channel, SocketAddress _addr) :
     call(* DatagramChannel.bind(..)) && target(_channel) && args(_addr);
   ...
}
```

Figure 13. A Code Snippet of *ResponderJoinPointTracker*

C. *Base Aspects*

The CommJ Infrastructure implements high-level IPC abstractions as base aspects, which fall into two categories, i.e., Communication aspects and Connection aspects. They

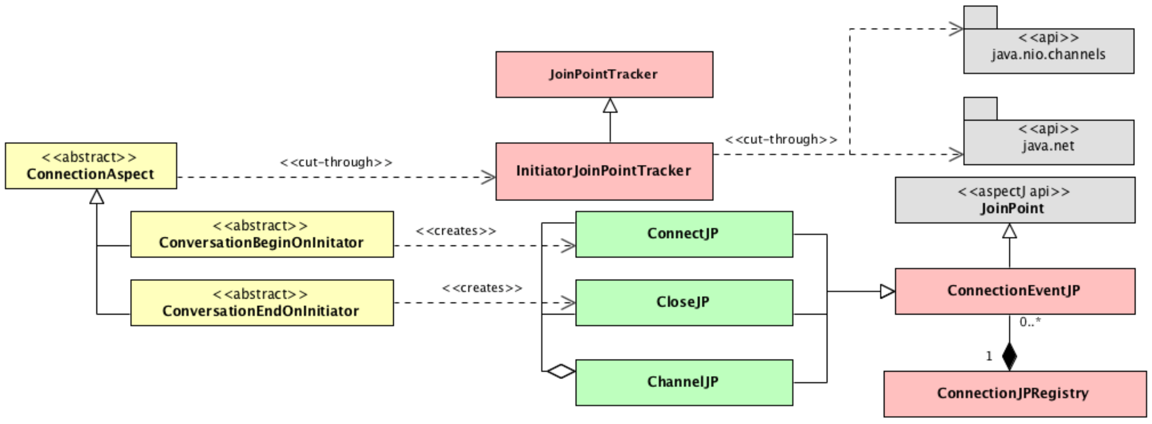


Figure 14. Connection Joinpoint and Base Aspects

```
public aspect InitiatorJoinPointTracker {

  private pointcut SocketConnectStyle1():
    call(Socket.new());

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

  private pointcut SocketConnectStyle4(String _host, int _port):
    call(Socket.new(String, int))  && args(_host, _port);

  private pointcut SocketConnectStyle5(Socket _socket, InetSocketAddress _endPoint):
    call(void Socket+.connect(SocketAddress)) && target(_socket) && args(_endPoint);

  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 15. A Code Snippet of *InitiatorJoinPointTracker*

cut through their respective joinpoint trackers and provide communication-related crosscutting concerns.

1) Message Aspects

All communication aspects are ultimately derived from the abstract Message Aspect class, which provides concrete pointcuts that dynamically track send and receive events (see Figure 16).

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

Figure 16. Pointcuts in *MessageAspect*

It is important to note that these pointcuts take CommJ joinpoint objects as parameters, because this is how advice is woven into these pointcuts, and can access conversation contexts.

The four specializations of Message Aspect correspond to four different kinds of conversation 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 17).

One way receive (OWR). An OWR conversation for a responder involves only one receive event. The conversation automatically ends for the responder after a receive event (see Figure 18).

Bi-directional (Request/Reply style of Conversation). Bi-directional conversations require a successful round-trip of a send and receive events. An RR Conversation Aspect, which applies to bi-directional conversations, defines pointcuts Start Conversation and End Conversation. The Start Conversation creates a Request Reply Conversation JP and starts a conversation when a sender invokes a sent event, the End Conversation retrieves the matching Request Reply Conversation JP from the Message JP Registry and ends a conversation when a Receiver invokes a receive event (see Figure 19 for more details).

Multi-step Conversation. It 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.

We implemented the multi-step conversation aspect (see Figure 20) by deriving from Message Aspect class and thereby inheriting the Message Send and Message Receive

```
public abstract aspect OneWaySendAspect
                        extends MessageAspect {
    public pointcut ConversationBegin(SendEventJP jp)....
}
```

Figure 17. OneWaySend aspect in RAL

```
public abstract aspect OneWayReceiveAspect
                        extends MessageAspect {
    public pointcut ConversationEnd(ReceiveEventJP jp)....
}
```

Figure 18. OneWayReceive aspect in RAL

```
public abstract aspect RRConversationAspect
                        extends MessageAspect {
    public pointcut ConversationBegin(RRConversationJP jp) ....
    public pointcut ConversationEnd(RRConversationJP jp) ....
              ....
}
```

Figure 19. RRConversation aspect in RAL

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

Figure 20. MultistepConversation aspect in RAL

pointcuts. A multistep conversation retrieves message, role, protocol and conversation information from Message class and creates a state machine instance if it does not already exist. During one application session, an aspect may apply several concurrent conversations for one type of state machine (i.e., a protocol as it applies to one role). The context for each conversation is maintained in terms of its own current state and associated state machine instance. In general, there are two types of state machines. Mealy and Moor state machines [19]. 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 21 and code snippet is in Figure 22. A CommJ state machine has two components: State and Transition. A State encapsulates the state name, whether it is in initial or final state, and its list of transitions. Transition is defined using four basic elements: Action Type, Message Type, From State, and To State. The Action Type is transition trigger and can be either a send or receive action. The Message Type is a filter or guard that specifies what types of messages may trigger the transition. From State defines the state before transition and To State defines the target state after transition.

When an application is loaded into memory, all application-level state machine classes are initialized and stored in State Machine Types - a hash map between application classes and state machine types. The Register methods, declared in abstract state machine and implemented by each application-level state machine, are called when applications are loaded through a static initialization block.

2) Connection Aspects

A Connection Aspect derives from a CommJ base aspect, which crosscuts Responder Joinpoint Tracker and Initiator Joinpoint Tracker pointcuts. The base connection aspect defines the following four pointcuts (see Figure 22):

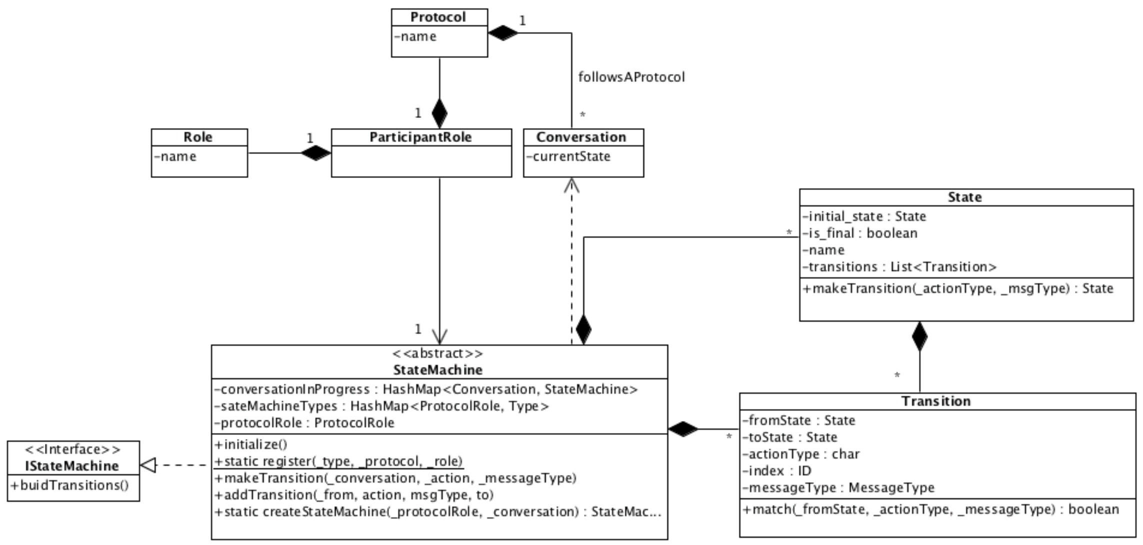
Connect pointcut. It crosscuts Initiator Joinpoint Tracker connection related pointcut and provides Connect pointcut.

Accept pointcut. It crosscuts Responder Joinpoint Tracker accept related pointcuts and provides an Accept pointcut.

Close Server pointcut. It crosscuts Responder Joinpoint Tracker and provides a Close Server pointcut.

Close Client pointcut. It crosscuts Initiator Joinpoint Tracker “close connection” pointcuts and provides Close Client pointcut.

Complete Connection Conversation. It inherits from Connection Aspect (Figure 23) and defines following



Protocol
-name
1
followsAProtocol
*
1
Role
-name
1
ParticipantRole
Conversation
-currentState
State
-initial_state : State
-is_final : boolean
-name
-transitions : List<Transition>
+makeTransition(_actionType, _msgType) : State
*
1
<<abstract>>
StateMachine
-conversationInProgress : HashMap<Conversation, StateMachine>
-sateMachineTypes : HashMap<ProtocolRole, Type>
-protocolRole : ProtocolRole
+initialize()
+static register(_type, _protocol, _role)
+makeTransition(_conversation, _action, _messageType)
+addTransition(_from, action, msgType, to)
+static createStateMachine(_protocolRole, _conversation) : StateMac...
*
*
Transition
-fromState : State
-toState : State
-actionType : char
-index : ID
-messageType : MessageType
+match(_fromState, _actionType, _messageType) : boolean
<<Interface>>
IStateMachine
+buidTransitions()

Figure 21. Design of Multi-step State Machine

```
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 ResponderJoinPointTracker.ChannelAccept(..))
             && args(_connectJp);

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

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

Figure 22. A Code Snippet of *Connection* Aspect

```
public abstract aspect CompleteConnectionAspect extends ConnectionAspect {

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

  public pointcut ConversationEndOnResponder(ChannelJP _channelJp) :
    execution(* CompleteConnectionAspect.EndResponder(ChannelJP)) &&
             args(_channelJp);

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

  ...
}
```

Figure 23. A Code Snippet of *Complete Connection* Aspect

```
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 24. A code snippet of *Total Turn Around Time Monitor*

pointcuts that help programmers to define conversations for total connection time on both responder and initiator sides.

The Conversation Begin On Initiator and Conversation End On Initiator pointcuts crosscut the state of request to establish and end a connection on the initiator.

The Conversation Begin On Responder and Conversation End On Responder pointcuts mark the start and end of connection related conversation on the responder.

We also define a helping initialization aspect, which loads application specific state machines and introduces conversation, role, protocol and message identity information before the application sends or receives any messages.

D. *Re-usable Aspect 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. Figure 24 shows part of the implementation of first one, i.e., Total Turn Around Time Monitor. Note how the advice in this aspect follows the Template Method pattern [12]. 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.

E. *Application-level Aspects*

This section provides four examples of communication and connection related crosscutting concerns implemented with CommJ.

1) *Measure Performance in Multi-step Conversation Process*

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 from *B*, it sends a message 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 25 shows a finite state machine for the *A Process Role* of this protocol. The behaviors for *B* and *C Process Roles* are considerably simpler and are shown in Figures 26 and 27, respectively.

The CommJ State Machine class includes a Build Transitions method that allows developers to define state machines in terms of states and message-event transitions. Figure 28 shows the implementation of this method to define a State Machine for the A Process Role

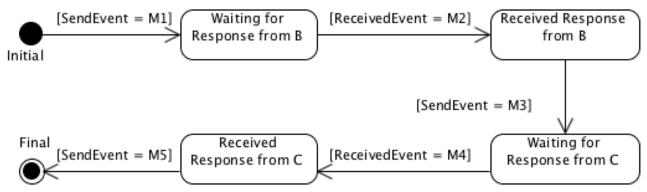


Figure 25. State Machine for the *A ProcessRole*

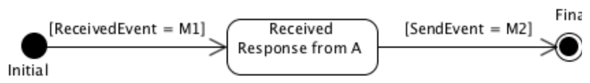


Figure 26. State Machine for the *B ProcessRole*

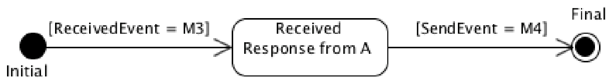


Figure 27. State Machine for the *C ProcessRole*

```
public aspect 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 28. State Machine Configuration for ProcessRoleA

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 conversations start times to conversations end times. The rolling window keeps track of these statistics for the current minute and the 10 previous minutes. Figure 29 shows the key pieces of code for an aspect that implement this performance measure crosscutting concern.

First notice how the aspect is derived from Total Turn Around Time Aspect 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.

2) *Version Control Aspect*

This example discusses the design and implementation of an aspect that can coordinate communications when different processes are following different versions 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 process A is following the updated

```
public aspect MyAppPerformanceMonitor
          extends TotalTurnAroundTimeMonitor {

 private ArrayList<Stats> statsList = new ArrayList(11);
 private int currentStatsIndex = 0;

 @Override
 public void End(MultistepConversationJP jp) {
   //Get number of elapsed minutes since begining 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 avgTurnaround;
   public Stats() {Reset(); }
   public void Reset() {
     startTime = currentTime;
     completeConvCount = 0;
     avgTurnaround = 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 turnaroundTime) {
     avgTurnaround = ((completeConvCount*avgTurnaround) +
       turnaroundTime)/(++completeConvCount);
   }
  }
}
```

Figure 29. Performance Measure Crosscutting Concern

```
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 30. Version Control Aspect for Messages Sent

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

Figure 31. Version Control Aspect for Messages Received

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.

The application-level version control aspects in Figures 30 and 31 extend RAL communication aspects discussed in Section IV.C. On sending the messages, One Way Send Aspect 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.

3) *Logging Responder and Initiator Connection Times for FTP*

This section describes aspects for logging responder and initiator connection times for the processes using FTP for file transfer. Assume that an FTP client establishes a TCP connection to an FTP server. 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.

As mentioned above, with FTP, there are two processes: an FTP client and FTP server. The server and client communicate using two messages, i.e., File Transfer Request and File Transfer Response. FTP client sends a File Transfer Request message to FTP server, after a connection has been established between the two processes. The File Transfer Request message contains the requested file name. When FTP server receives the request, it starts sending the response message (File Transfer Response) to the client, which includes the file information, data chunk number and its completion status. Following paragraphs describe related application-level aspects for initiator and responder.

Aspect - Logging Initiator Connection Time. This is an application-level connection aspect, developed using the RAL connection aspect, i.e., Complete Connection Aspect in Section IV.C. It logs the time between initiating connection request to the responder (FTP server) and ending of connection on the initiator (FTP client) using Conversation Begin On Initiator and Conversation End On Initiator pointcuts (see Figure 32).

Aspect - Logging Responder Connection Time. This is an application-level connection aspect, developed using RAL connection aspect, i.e., Complete Connection Aspect in Section IV.C. It logs the time period between acceptance of connection request from initiator and ending of connection on the responder using Conversation Begin On Responder and Conversation End On Responder pointcuts (see Figure 33).

V. Extended Quality Model

To measure the maintainability and reuse, we use Sant'Anna's Comparison Quality Model (CQM) [21] and extends it with new factors and internal attributes, forming the Extended Quality Model (EQM); see Figure 34. We use Sant'Anna's model because it is more generalized to measure different concerns of reuse and maintenance as compared to Lopes' work [22]. Additionally, this model is strong enough to be applied to different types of implementations, discussed in this paper.

Sant'Anna builds the CQM using Basili's General Quality Methodology (GQM) [23], which provides a three-step framework: (1) list the major goals of the empirical

```
public aspect InitiatorTimeAspect extends CompleteConnectionAspect {
  private long startTime = 0;
  static String timingInfo = "";

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

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

Figure 32. Third Code Snippet of TurnAroundTimeAspect

```
public aspect ResponderTimeAspect extends CompleteConnectionAspect{
  private long startTime = 0;
  static String timingInfo = "";

  Object around(ChannelJoinPoint _channelJp) : ConversationBeginOnResponder(_channelJp){
    startTime = Systems.currentTimeMillis();
    return proceed(_channelJp);
  }

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

Figure 33. Fourth Code Snippet of TurnAroundTimeAspect

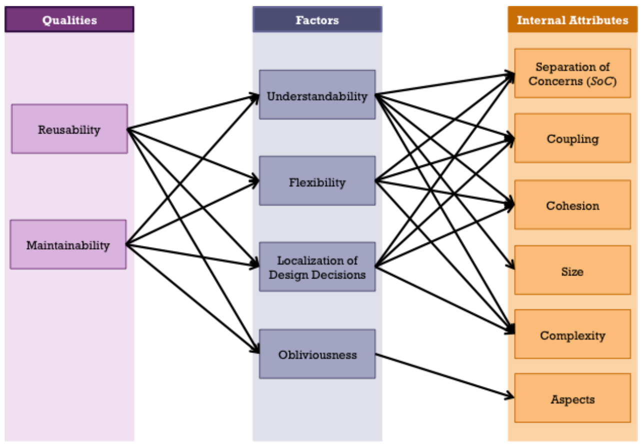


Figure 34. Extended Quality Model (*EQM)*

study, (2) derive from each goal the questions that must be answered to determine if the goals have been met, and (3) decide what must be measured to be able to answer the questions adequately.

Santa'Anna organized the CQM into four components: qualities, factors, attributes, and metrics. The qualities are the high level characteristics that we want to primarily observe in our software. Factors are the secondary quality attributes (more granular than qualities) that influence the defined primary attributes, i.e., qualities. 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. Finally, the metrics are ways to measure the attribute.

We also made a few enhancements in EQM and believe that doing so will further strengthen the model. First, our model creates a dependency of maintainability and reusability upon flexibility and understandability factors. Secondly, because our experiment involves crosscutting concerns, so we introduced two important missing factors, i.e., code obliviousness [24] and localization of design decisions [25]. Research and practice also validate that modular code is more maintainable [26] when obliviousness and localization of design decisions are present.

A. *Qualities*

Qualities are the highest level of abstractions in EQM. For our experiment with CommJ, we considered maintainability and reusability would be the two most important qualities to focus on.

- Reusability: Reusability exists for a given software element, when developers can use it for the construction of other elements or systems [27].
- Maintainability: Maintainability is the activity of modifying a software system after initial delivery [28]. It is the ease with which software components can be modified.

B. *Factors*

Following are the list of factors in our EQM.

- Understandability: indicates the level of difficulty for studying and understanding a system design and code.
- Flexibility: 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: 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 [29].
- Obliviousness: is a special form of low coupling wherein base application functionality has no dependencies on crosscutting concerns [21].

Localization of design decisions, and code obliviousness were not part of CQM. However, we introduced them into our EQM for two reasons. First, in his landmark paper [25], 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.

C. *Attributes*

Following are the internal attributes in our EQM.

- Separation of Concerns (SoC): defines ability to identify, encapsulate and manipulate those parts of software that are relevant to a particular concern.
- Coupling: 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: is a measure of the closeness of relationship among the internal components of a method, class, subsystem, etc.
- Size: represents the length of a software system's design and code.
- Complexity: characterizes how and how much components are structurally interrelated to one another.
- Tangling: exists when a single component includes functionality for two or more concerns, and those concerns could be reasonably separated into their own components.
- Scattering: 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.

D. Measurement Metrics

Figure 35 presents the metrics the EQM uses to measure each of the internal attributes. Detail descriptions of these metrics follow below.

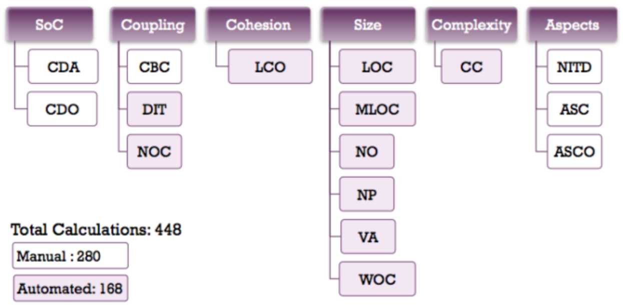


Figure 35. Measurement Metrics in *EQM*.

1) SoC 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 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.

2) Coupling Metrics

The EQM uses 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.

3) Cohesion Metrics

The EQM uses the Lack of Cohesion in Operations (LCO) for measuring cohesion and tangling among components.

Specifically, LCO measures the lack of cohesion of a class or aspect by looking at lines of code within method and advice pairs, which do not access the same instance variables. If the related methods do not access the same instance variable, they logically represent unrelated components and hence should be separated.

4) Complexity Metrics

McCabe's Cyclomatic Complexity (CC) [30] is the EQM's chosen metric for measuring complexity. 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.

5) Obliviousness Metrics

The EQM introduces the following new metrics for obliviousness metrics: Number of Inter-type Declarations (NITD), Aspect Scattering over Components (ASC), and 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.

6) Size Metrics

The EQM uses 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 reuse and maintenance. MLOC counts the method lines of code. Kremer [31] 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. Sometimes 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 [21] 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 [31] 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.

VI. Hypotheses

CommJ's theoretical foundation and design lead to the following seven hypotheses, with respect to comparing the reusability and maintainability of IPC software built with *CommJ* instead of just *AspectJ*.

- *Hypothesis 1:* If crosscutting IPC 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 V.D.1.) than equivalent systems developed with AOP design techniques.
- *Hypothesis 2:* If crosscutting IPC concerns are encapsulated in *CommJ* aspects, then the software has *lower coupling* (as described by CBC, DIT, NOC in Section V.D.2) than equivalent systems developed with AOP design techniques.
- *Hypothesis 3:* If crosscutting IPC concerns are encapsulated in *CommJ* aspects, then the software has *higher cohesion* and less tangling (as described by LCO in Section V.D.3. than equivalent systems developed with AOP design techniques.
- *Hypothesis 4:* If crosscutting IPC concerns are encapsulated in *CommJ* aspects, then the software is not significantly *complex* (as described by CC in Section V.D.4) than equivalent systems developed with AOP design techniques.
- *Hypothesis 5:* If crosscutting IPC concerns are encapsulated in *CommJ* aspects, then the software is significantly more *oblivious* (as described by NITD, ASC, ASCO in Section V.D.5) than equivalent systems developed with AOP design techniques.
- *Hypothesis 6:* If crosscutting IPC concerns are encapsulated in *CommJ* aspects, then the software is not significantly larger (as described by LOC, MLOC, NO, NP, VA, WOC in Section V.D.6) than equivalent systems developed with AOP design techniques.
- *Hypothesis 7:* If crosscutting communication concerns are encapsulated in *CommJ* aspects, then extension for new requirements touches fewer components or lines of code (as measured by Eclipse IDE diff function) than equivalent systems developed with AOP design techniques.

VII. Experiment Method

The following sections briefly describe the steps of a preliminary experiment that authors used to test the hypotheses.

A. *Experimental Planning and Approval*

In the first step, we developed a plan and submitted an application for conducting this Human Research Experiment to the IRB [32], and received approval. Each of us also had to pass an online human research experiment-training course offered through Collaborative Institutional Training Initiative (CITI) [33].

B. *Selection of Applications and Crosscutting Concerns*

We selected sample software applications (see Table I) that were multithreaded, distributed, and used either JDK sockets or channels for communications. The applications were diverse in the way they implemented IPC and therefore provide good coverage of different types of communication heterogeneities. Finally, each application supported more than one communication protocol.

Since the experiment would eventually require developers to modify or extend applications for requirements that represented communication-related crosscutting concerns, our methodology included a step, which systematically selected our representative crosscutting concerns. Developers would have to apply each of these to the applications, individually. Additionally, to minimize noise in our data, we wanted to make sure that these crosscutting concerns were sufficiently simple that novice programmers could understand them and come up with a

TABLE I. SELECTED SAMPLE APPLICATIONS

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

TABLE II. 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
*Measuring Performance***	It measured some performance related statistics for message-based communications between a sender and receiver
*Symmetric-Key Encryption***	It encrypted the communication between a sender and receiver using symmetric-key encryption
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
MessageLoggingByConversation	Log messages by conversations in a developer-defined format and repository

** Selected cross-cutting concerns for the experiment

solutions in less than 10 hours. We also come up with a list of possible crossing concerns that the subject programmers would have to implement in the applications (See Table II). Those marked with "**" represent the ones selected for the experiment.

C. Recruitment and Training of Participants

To transparently recruit the candidates, we sent invitation letters and recruited seven volunteer developers who met the participation criteria, specifically they were experienced in object-oriented software development, Java, and with software-engineering design principles, such as modularity and reusability. We then randomly organized them into two study groups: A and B. Group A would use AspectJ only and Group B would use CommJ on top of AspectJ. Next, the participants completed a survey that assessed their background and skill levels. We also provided AOP training to developers in Group A, and worked through some practice applications with them. Similarly, we trained Group B developers in CommJ, and worked through some practice applications with them.

D. Experiment Phases

In the first phase, participants filled a pre-implementation questionnaire, developed the application using initial requirements, recorded hourly journals and completed a post implementation questionnaire. In the second phase, we requested enhancements (sample applications and crosscutting concerns), had them revised their implementation accordingly, and then collected those software systems. Participants again completed the pre and post questionnaire and wrote their experiences in the hourly journals.

Finally, after the second phase, we analyzed and evaluated the reusability and maintainability using various software artifacts, which included surveys, questionnaires, hourly journals, and actual code.

We used both manual computation and automated tools to compute measurements for all 16 metrics. Experiment generated a total of 28 software systems. With 16 code metrics in the EQM, we had a total of 448 measurements, 280 computed automatically with a tool [34] and 168 calculated manually.

VIII. EXPERIMENT RESULTS AND CONCLUSIONS

This section presents the data collected from the experiment and our results in context of the seven hypotheses. In the following graphs, the vertical axes represent the measurements, and the horizontal axes represent the activities of the experiment. For each activity there are two bars: a blue bar is for the results of AspectJ-only group and a green bar for CommJ group.

A. 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 V.D.1. for details on metrics). We found CDA and CDO did decrease for the CommJ group. In Figure 36, the vertical axes represent the CDA and CDO measurements, and the horizontal axes represent the four activities of the experiment.

Not only were CDA and CDO values reduced using CommJ, but they were 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 concern were significantly more diffused in the application because the pointcuts had to be tied to programming constructs instead of communication abstractions.

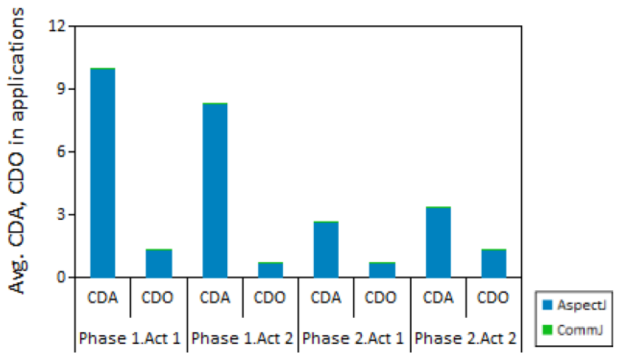


Figure 36. CDA, CDO coverage over phases.

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

B. *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 V.D.2. for details on metrics). Figure 37 indicates 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.

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 conclude that Hypothesis#2 holds true for reduced coupling in CommJ than in AspectJ.

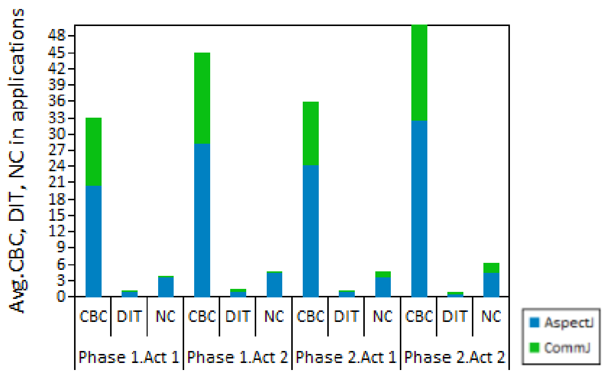


Figure 37. CBC, DIT, NC coverage over phases.

C. *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 V.D.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 38 demonstrates that CommJ maintains a lower value for LCO than AspectJ in all four phases of the experiment. Sant'Anna [21] says that LCO measures the degree to which a component implements a single logical function. These results indicate that CommJ implementations were more cohesive and logical than AspectJ, hence have a lower LCO value. Therefore, we conclude that Hypothesis #3 holds true for increased cohesion in CommJ than in AspectJ.

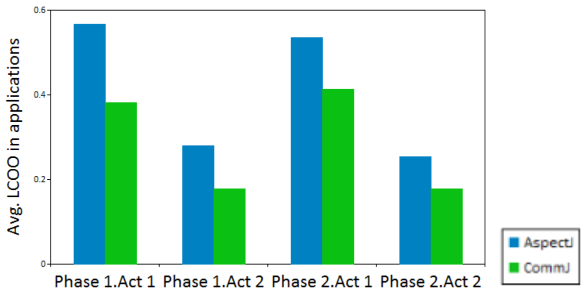


Figure 38. LCO coverage over phases.

D. *Complexity*

Hypothesis #4 theorized that if crosscutting communication concerns are effectively encapsulated in CommJ aspects, the software is significantly less complex (as described by CC in Section V.D.4.) than equivalent systems developed with AOP design techniques. In other words, the CC value for CommJ should be less than AspectJ. Figure 39 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 conclude that Hypothesis #4 holds true for less complex software in CommJ than AspectJ.

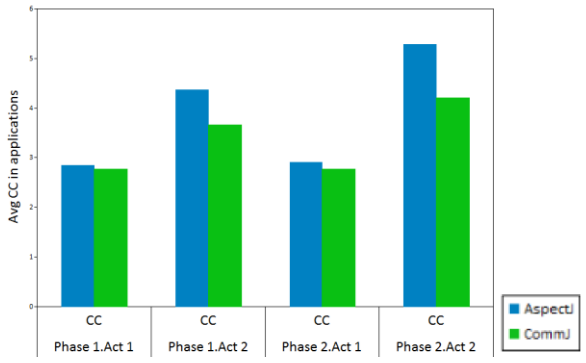


Figure 39. CC coverage over phases.

E. *Obliviousness*

Hypothesis #5 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 V.D.5.) than equivalent systems developed with AOP design techniques. In other words, NITD, ASC, ASCO for CommJ should be less than AspectJ. Figure 40 shows that CommJ implementations significantly reduced the values of NITD, ASC and ASCO metrics.

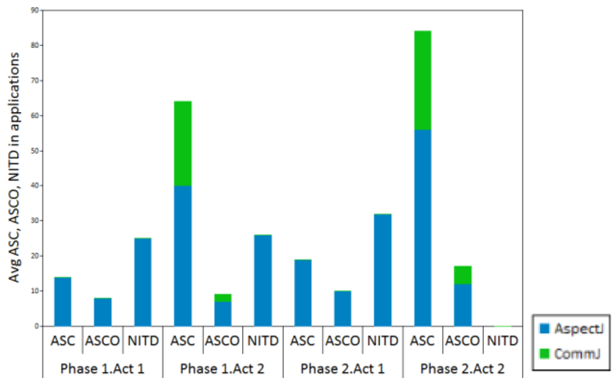


Figure 40. ASC, ASCO, NITD coverage over phases.

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 between the application and aspect components, which CommJ provides but are missing in AspectJ. Therefore, we conclude that Hypothesis #5 holds true for less oblivious software concerns in CommJ than AspectJ.

F. *Reduced Size*

Hypothesis #6 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 V.D.6.) 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. Figure 41 shows 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 abstractions using UMC (Section III.A.) 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 41 gives 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. Thus, from these results, we can conclude that Hypothesis#6 holds true for improved code size in CommJ than in AspectJ.

G. *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 (as measured by CR, CM as follow) than equivalent systems developed with AOP design techniques, where CR and CM are as follows:

- *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.

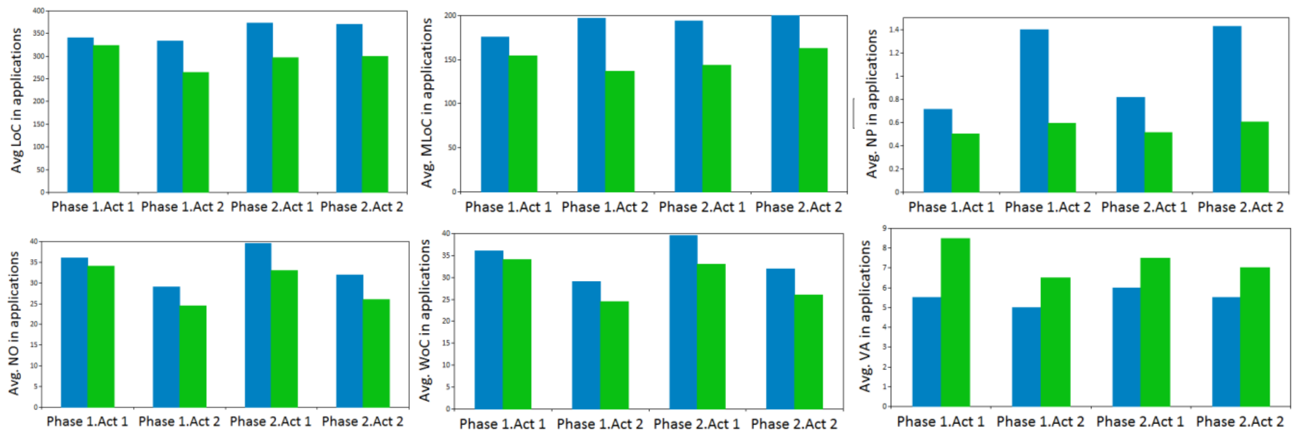


Figure 41. LoC, MLoC, NP, NO, WoC coverage over phases.

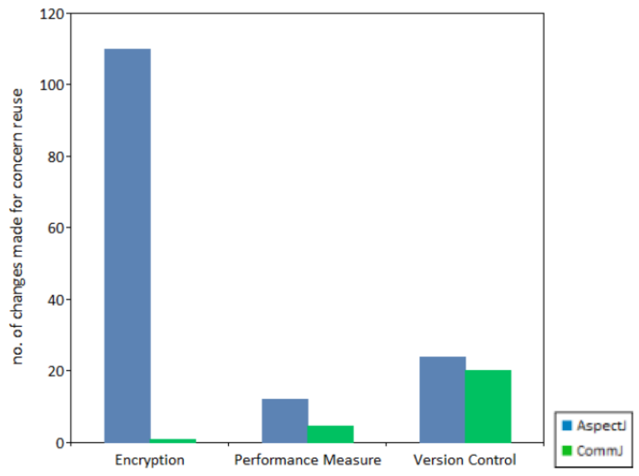


Figure 42. CR over Extensions

In other words, CR, CM values for CommJ should be less than AspectJ. From the results shown in Figure 42, 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 43 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 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 VIII.A above.

Figure 44 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.

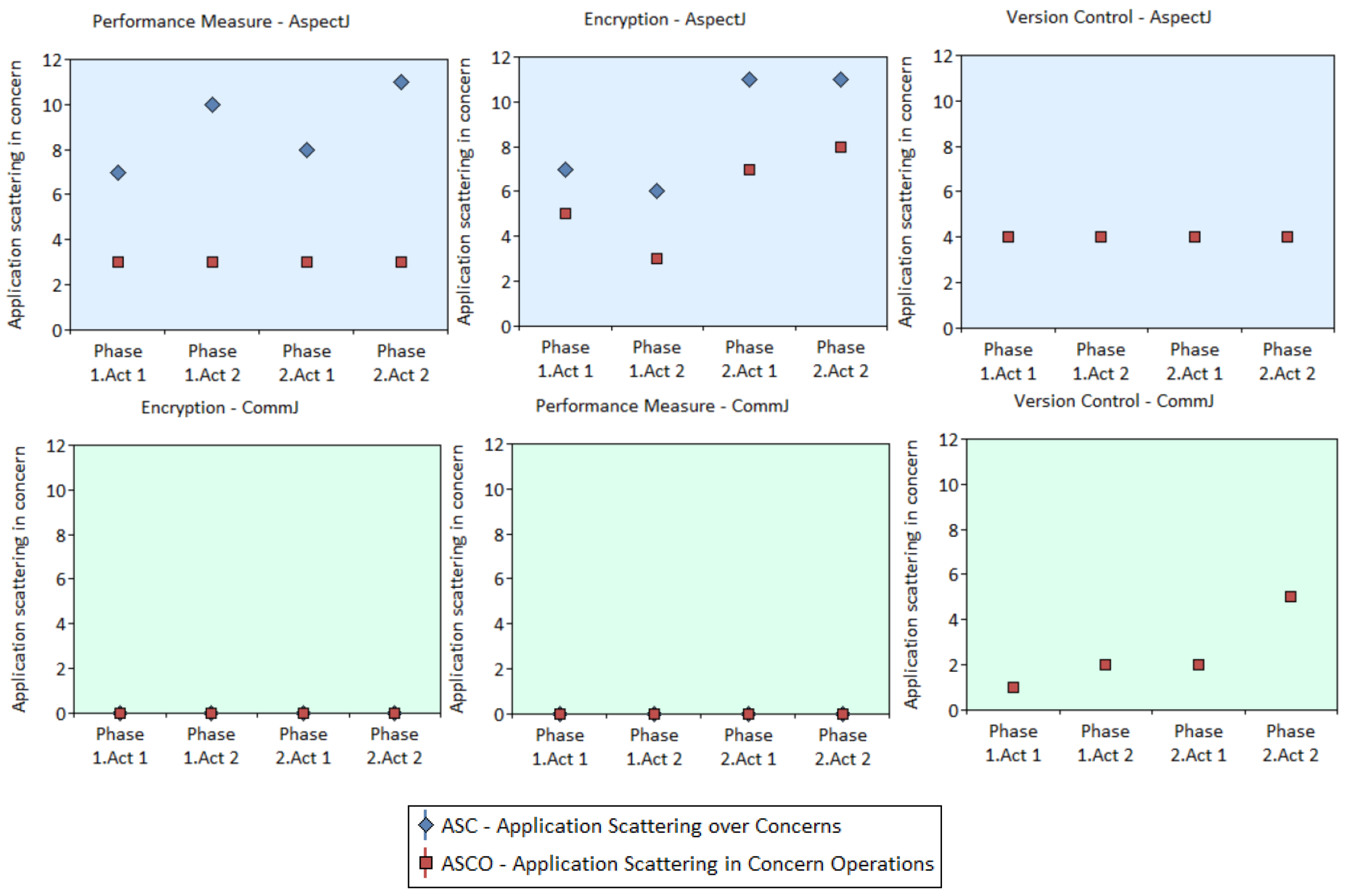


Figure 43. ASC and ASCO over Phases in AspectJ and CommJ

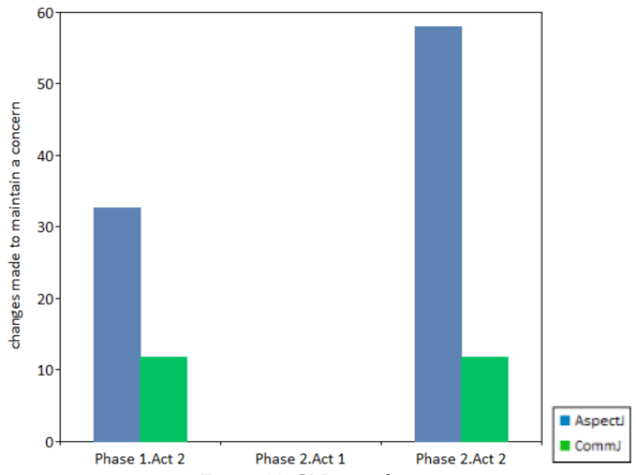


Figure 44. CM over phases

Figure 45 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 and NITD metrics results respectively. The points in the above graph 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 Section VIII.A and Section VIII.E, respectively.

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

H. *Other Useful Observations*

Besides analysis of the hypotheses, we also collected a handful observations from participants' questionnaires 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

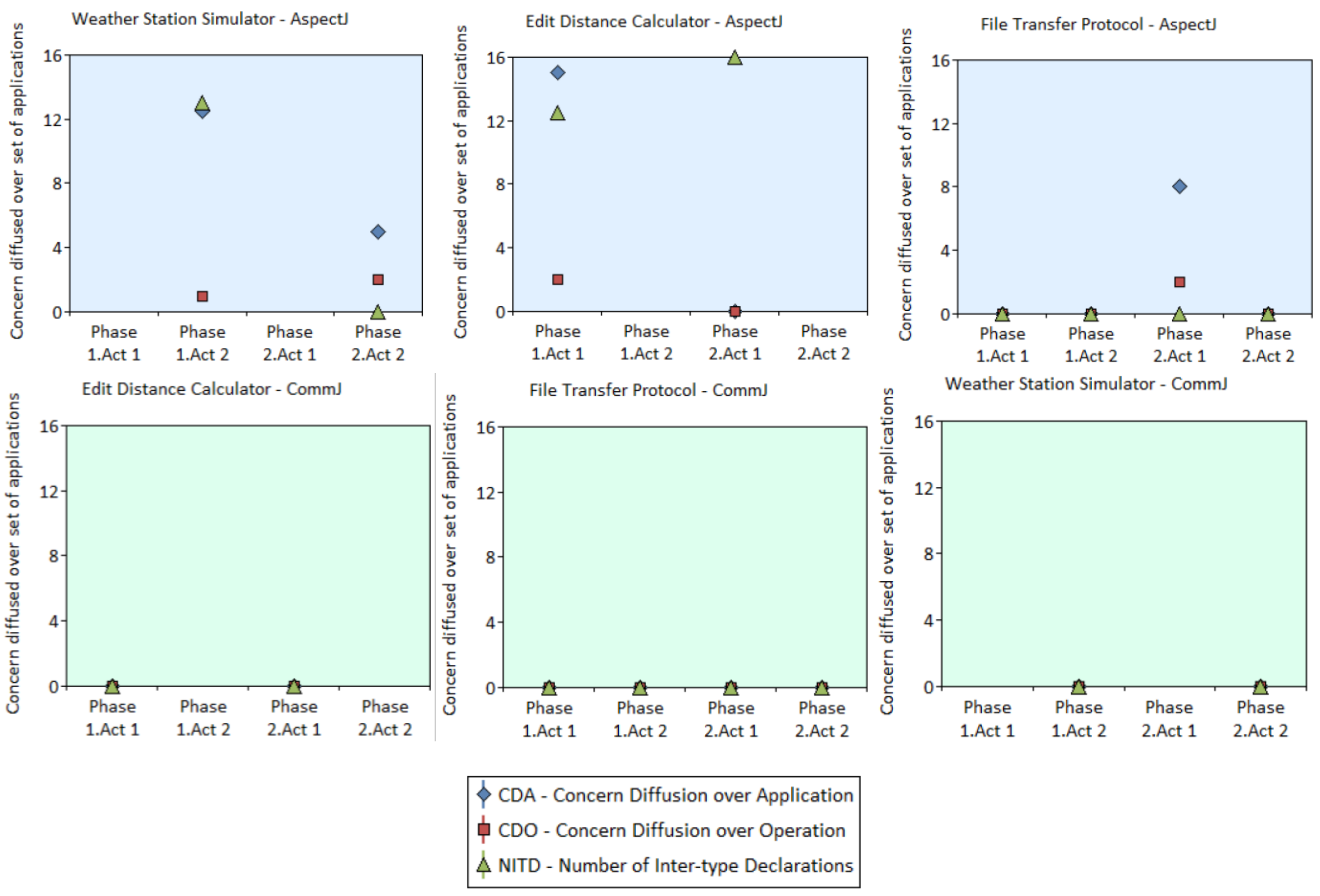


Figure 45. CDA, CDO and NITD in AspectJ and CommJ

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 introduced any 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.

IX. Summary and Future Work

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. Findings from this initial experiment revealed that crosscutting concerns programmed in CommJ delivered more modular, reusable and maintainable programs. We hope to pursue larger and varied software-engineering productivity experiments to verify this belief.

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 extend 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] A. Raza, S. Clyde, and J. Edison, "Communication Aspects with CommJ: Initial Experiment Show Promising Improvements in Reusability and Maintainability," In ICSEA 2014, Nice, France.

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

[3] L.D. Benavides Navarro et al., "Invasive patterns for distributed programs," In OTM Confederated Int. Conf., Vilamoura, Portugal, 2007, pp. 772-789.

[4] G. Kiczales et al., "Aspect-oriented programming," (ECOOP), 1997, pp. 220-242.

[5] ApectJ, http://www.eclipse.org/aspectj/, last updated on May 13, 2016.

[6] AspectWorkz2, http://aspectwerkz.codehaus.org/ss, last updated on May 13, 2016.

[7] JBoss AOP, http://www.jboss.org/jbossaop, last updated on May 13, 2016.

[8] Spring AOP, org.springframework, last updated on May 13, 2016.

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

[10] 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.

[11] C. Sant'Anna, A. Garcia, C. Chavez, C. Lucena, and A. Von Staa, "On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework," in 17th Brazilian Symposium on Software Engineering (SEES 2003), Manaus, Brazil (2003), PUC-RioInf.MCC26/03.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 1995.

[13] 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.

[14] R. Douence, D.L. Botlan, J. Noye, and M. Sudholt, "Concurrent aspects," In. Proc. 5th Int. Conf. GPCE, Portland, OR, 2006, pp. 79-88.

[15] 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.

[16] P. Tarr, H. Ossher, W. Harrison, and S.m. Sutton, "N degrees of separation: Multi-dimensional separation of concerns," In Proc.21st Int. Conf. Software Engineering, Los Angeles, CA, 1999, pp. 107-119.

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

[18] 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.

[19] S. Chiba, “Load-time structural reflection in Java,” In Proc.14th ECOOP, Cannes, France, 2000, pp. 313-336.

[20] T.J. Brown, I. Spence, and P. Kilpatrick, “Mixin programming in Java with reflection and dynamic invocation,” In Proc. 2nd Workshop on Intermediate Representation Engineering for Virtual Machines, Dublin, Ireland, 2002, pp. 25-34.

[21] C. Sant’Anna, A. Garcia, C. Chavez, C. Lucena, and A. Staa, “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.

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

[23] R. Basili, G. Caldiera, and H. Rombach, “The goal question metric approach,” In Encyclopedia of Software Engineering, vol. 2, J.J. Marciniak, Ed. Hoboken, NJ: Wiley, 1994, pp. 528-532.

[24] 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.

[25] 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.

[26] 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.

[27] J. McCall, P.K. Richards, and G.F. Walters, “Factors in software quality,” NTIS, Alexandria, VA, Tech. Rep. AD-A049-014, 015, 055, 1977.

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

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

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

[31] 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.

[32] Institutional Review Board (IRB), http://rgs.usu.edu/irb, retrieved: Mayss 13, 2016.

[33] Collaborative Institutional Trainig (CIIT), https://www.citiprogram.org, retrieved: May 13, 2016.

[34] Metrics plugin, http://metrics2.sourceforge.net, retrieved: May 13, 2016.