5-2015

Improving Reuse of Distributed Transaction Software with Transaction-Aware Aspects

Anas Ahmad AlSobeh
Utah State University

IMPROVING REUSE OF DISTRIBUTED TRANSACTION SOFTWARE WITH TRANSACTION-AWARE ASPECTS

By

Anas “Mohammad Ramadan” Ahmad AlSobeh

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

of

DOCTOR OF PHILOSOPHY

in

Computer Science

Approved:

UTAH STATE UNIVERSITY
Logan, Utah

2015

ABSTRACT

Improving Reuse of Distributed Transaction Software with Transaction-aware Aspects

by

Anas AlSobeh, Doctor of Philosophy

Utah State University, 2015

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

Implementing crosscutting concerns for transactions is difficult, even using Aspect-Oriented Programming Languages (AOPLs) such as AspectJ. Many of these challenges arise because the context of a transaction-related crosscutting concern consists of loosely-coupled abstractions like dynamically-generated identifiers, timestamps, and tentative value sets of distributed resources. Current AOPLs do not provide joinpoints and pointcuts for weaving advice into high-level abstractions or contexts, like transaction contexts. Other challenges stem from the essential complexity in the nature of the data, operations on the data, or the volume of data, and accidental complexity comes from the way that the problem is being solved, even using common transaction frameworks. This dissertation describes an extension to AspectJ, called *TransJ*, with which developers can implement transaction-related crosscutting concerns in cohesive and loosely-coupled aspects. It also presents a preliminary experiment that provides evidence of improvement in reusability without sacrificing the performance of applications requiring essential transactions. This empirical study is conducted using the extended-quality model for transactional application to define measurements on the transaction software systems. This quality model defines three goals: the first relates to code quality (in terms of its reusability); the second to software performance; and the third concerns software development efficiency. Results from this study show that *TransJ* can improve the reusability while maintaining performance of *TransJ* applications requiring transaction for all eight areas addressed by the hypotheses: better encapsulation and separation of concern; loose Coupling, higher-cohesion and less tangling; improving obliviousness; preserving the software efficiency; improving extensibility; and hasten the development process. (180 pages)

PUBLIC ABSTRACT

Improving Reuse of Distributed Transaction Software with Transaction-aware Aspects

Anas AlSobeh

Distributed transaction processing systems (DTPSs) 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. Aspect orientation has the potential of reducing this kind of complexity; however, aspect-oriented programming languages and frameworks currently only allow weaving of advice into contexts derived from traditional executable structures. This dissertation introduces an abstract independent framework, called *TransJ,* for weaving crosscutting concerns into distributed transactions, which are high-level runtime abstractions. *TransJ* is an extension of AspectJ that defines interesting joinpoints relative to transaction execution and context data for woven advice. Specifically, it implements distributed transaction processing system concepts related to a) transactions in general, b) the kinds of information that comprise their contexts, and c) events that represent interesting timepoints/places for when/where the crosscutting concerns might augment an application's core function or the underlying transaction processing system. We capture these concepts in a conceptual model, called *Unified Model for Joinpoints Distributed Transactions* (UMJDT). This dissertation presents preliminary evidence that the advice of weaving can reduce complexity of cross-cutting concerns in software systems with distributed transaction. Specifically, the results show eight different ways in which TransJ can improve the reuse with preserving the performance of applications requiring transactions. Informally, these hypotheses are that *TransJ* yields 1) better encapsulation and separation of concern; 2) looser coupling and less scattering; 3) higher cohesion and less tangling; 4) reduces complexity; 5) improves obliviousness; 6) preserves efficiency; 7) improves extensibility; and 8) hastens the development process.

Also, this dissertation suggests further research for studying the modularity, abstractions, comprehensibility, and flexibility achieved through the ability to weave a crosscutting concerns into transactions directly.

ACKNOWLEDGMENTS

It is a pleasure to thank the many people who made this dissertation possible. Initially, I would like to show gratitude and thanks my advisor, Dr. Stephen Clyde, who led my steps in this cumulative process. I could not imagine having a better supervisor for my Ph.D. He provided invaluable assistance, encouragement, good teaching, lots of good ideas, and support throughout the entire process. Definitely, writing this dissertation would have been extremely tough without his presence.

Very special thanks go to my committee members, Dr. Xiaojun Qi, Dr. Curtis Dyreson, Dr. Nicholas Flann and Dr. Bedri Cetiner for giving me the extra strength and motivation to get things done.

Lastly, and most importantly, I offer special thanks to my family: my mother “Nayma’a AlSababha”, my father “Mohammad Ramadan AlSubh”, my wife “Amani Shatnawi”, my son “Omar”, and my daughter “Mayr” for their encouragement, moral support, and patience. They lighted my life, taught me, supported me, and loved me. I could not have done it without all of you. To them I dedicate this dissertation.

Anas AlSobeh

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Distributed Transaction Processing Systems (DTPSs) facilitate transactions that span heterogeneous, shared resources in distributed-computing environments. Often an implementation of such a system executes on many machines separated by physical distances ranging from a few inches to thousands of miles. A challenge with the implementation of DTPS is that some properties and functionalities cannot be easily encapsulated and localized into loosely coupled abstractions, which increases the complexity of the system.

Frederick Brooks characterizes software complexity as either essential or accidental, where essential complexity stems from the very nature of the problem being solved by the software and accidental complexity comes from the way that the problem is being solved [1]. A DTPS may have essential complexity in the nature of the data, operations on the data, or the volume of data. However, issues such as logging, persistence, resource location, and even distribution itself are more likely to be sources of accidental complexity, because they are not usually inherent parts of the problem. When these issues are secondary to the primary purposes of a DTPS, it is common to find logic for them scattered throughout the software and tangled into core application logic (see Chapter 2). For example, similar snippets of code for concurrency-control operations, like locking and unlocking, may occur throughout the system, even though they all implement the same basic concerns or requirements.

Object-oriented Programming (OOP) refers to a programming paradigm characterized by the concepts of abstraction, encapsulation, sub-typing, information hiding, inheritance, and polymorphism. It encourages the decomposition of real world problems into objects that encapsulate the attributes and behavior in meaning abstractions. OOP has proven to be effective in modeling common hierarchical behaviors, but falls short in modeling behaviors that span (i.e., crosscut) multiple unrelated modules (i.e., contexts) [2]. Attempts to implement such crosscutting concerns in OOP often result in systems that are difficult to reuse or maintain: this is where *Aspect-Oriented Programming* (AOP) comes in [3]. AOP and OOP are not competing programming approaches, but in reality complement each other perfectly.

AOP can help manage both essential and accidental complexity by localizing and encapsulating crosscutting concerns in first-class software components, called *aspects* [3]. An aspect is very much like a class in OOP and an aspect instance is like an object, except that an aspect defines special methods, called *advices*, which are automatically woven into the core application according to specifications, called *pointcuts*. A pointcut identifies a set of *joinpoints* – a logical intervals in the execution flow of the system where and when weaving of advice takes place. Each joinpoint begins and ends relative to static places in the source code, called *shadows* [4]. Weaving is the process of composing core functionality modules with aspects, thereby yielding a working system [3].

One could argue that a good programmer can do the same thing in OOP by defining classes for the crosscutting concerns and hard coding calls to methods of those classes in all the right places. However, the issue is not whether it can be done; rather, it is the difference in abstractions. AOP offers better abstractions for separating crosscutting concerns from core functionality that do require core functionality to dependent on crosscutting concerns in any way. An *Aspect-Oriented* (AO) developer should be able to add/remove aspects to/from a project without changes to any other code. Some authors refer to this as a principle, called *obliviousness* [6][84].

In 1997, Kiczales et al. designed AspectJ as a compatible extension to the Java Programming Language (JPL) to encapsulate crosscutting concerns that would otherwise lead to *scattering* and *tangling*[1] [7]. It allows application programmers to weave advice for the logic of crosscutting concerns into joinpoints that correspond to constructor calls/executions, method calls/executions, class attribute references, and exceptions [8][9][10]. AspectJ provides separate mechanisms for defining an aspect and identifying its interaction with an underlying core software system [33]. Therefore, it enables developers to modify and extend aspect modules independently from the system modules into which they are woven. In many cases, it can make an implementation clearer while requiring less code [12][13]. This enhances developers' ability to express the separation of concerns necessary for a well-designed, maintainable software system [5][29]. An *Eclipse-AspectJ Development Tools* (AJDT) is an environment that has been used as an open-source project-enhanced IDE support for *Aspect-Oriented Software Development* (AOSD) with Java and AspectJ [8][14].

[1] Scattering occurs when a concern/functionality is assigned to several components, where the assigned concern is unrelated to the key concepts the components model in the application domain. Tangling occurs when code implementing a crosscutting concern is mixed with code implementing primary functionality of a component [50] [51].

However, AspectJ, like many other existing *Aspect-Oriented Programming Languages* (AOPLs) and frameworks, suffers from the lack of capabilities that would handle high-level runtime abstractions; therefore, it only allows the weaving of advice into the execution of code-based contexts, such as methods, constructors, fields, and exceptions. They do not directly allow behaviors to be woven into more abstract contexts, such as transactions. The transaction represents a major crosscutting concern in DTPSs because it is difficult to modularize with current technologies. Furthermore, few approaches have investigated the use of aspects for distributed transaction programming. Once we identified the weakness in AOP for weaving advice into distributed transactions, we elaborated on the problem from different dimensions (see Chapter 2) and reviewed the related literature (see Chapter 11).

As foreseeable of any new technology, the AOP developers continue to apply AspectJ in its respective domains and propose new AOPL technologies and frameworks to address their needs. Yet, these proposed features are not adaptable to implementing concerns with respect to high-level abstractions, like transactions. A *transaction* is a set of operations on shared resources, such that its execution results in either the successful completion of all operations or the completion of no operation. Besides this all-or-nothing property, called *atomicity*, transactions are *consistent*, *isolated*, and *durable*, meaning that persistent data will only change from one valid state to another; that other concurrent transactions cannot see the effects of a transaction until it is completed; and, that effects of a transaction become persistent after completion, even if there is system failure. Together, *Atomicity*, *Consistency*, *Isolation*, and *Durability* are often referred to as the ACID properties [15][16].

Even though transactions core concepts in many distributed systems, they are rarely treated as a first-class programming concept. Consequentially, the logic for transactions is, in general, scattered or spread across several units of the DTPS [16][17]. Thus, when changes occur to that logic, there can be a large ripple effect on the whole system.

Distributed transactions are transactions, but their operations execute on multiple host machines, ideally with improved throughput. From a logical perspective, a distributed transaction can be a flat sequence of operations or a hierarchy of sub-transactions, also known as *nested transactions*. In the latter case, nested transactions may execute concurrently and sequentially [39].

Regardless of whether a distributed transaction is a flat or nested transactions, it is an ephemeral concept that spans multiple execution threads and operations and may use a variety of distributed resources. Therefore, from an execution-timeline perspective, it may seem non-contiguous and unevenly spread out. A transaction's context is not tied to code constructs, like constructors and methods, in a single thread of execution; rather, it consists of loosely coupled abstractions like dynamically generated identifiers, timestamps, and tentative value sets for distributed resources. This makes it very difficult for aspect-oriented developers to localize and encapsulate crosscutting concerns that apply to transactions as execution units.

It is important to AOP's long-term success to allow the Distributed Transaction Application (DTA) programmer to implement crosscutting transaction concerns directly and effectively, while improving reusability, and modularity without sacrificing performance. This dissertation investigates how transactions can be provided to application programmers at the programming language level so programmers can weave transaction-related crosscutting concern into a DTPS in a productive and reusable way, while preserving performance, core functionality and obliviousness to crosscutting concerns.

To accomplish this overall objective, this dissertation has two main contributions. First, it provides a foundation for developing an extension to AspectJ, called *TransJ*, that allows application programmers to weave aspect behaviors for transaction-related crosscutting concerns into such joinpoints. *TransJ* offers a framework, independent the underlying transaction framework that allows Aspect-Oriented developers to treat transactions as first-class concepts into which compilers can weave transaction-related crosscutting concerns. Specifically, it defines interesting time points/places for when/where the crosscutting concerns might augment an application's core functional or the underlying transaction processing system. To establish this extension, we captured key transaction events and context information in a conceptual model, called *Unified Model for Joinpoints of Distributed Transactions* (UMJDT) [18]. This model defines interesting joinpoints relative to transaction execution and context data for woven advice. The implementation of *TransJ* included an implementation of UMJDT that provided the ability to weave advice into transaction program execution before, after, or around complete transactions or individual transaction operations. The implementation perspective of *TransJ* utilizes the *Java Transaction API* (JTA) standards, which is the de facto standard in the Sun *Java Enterprise Edition* (J2EE) for handling distributed transaction development today. Chapters 3 and 4 provide a high-level explanation of *TransJ* architecture, including the

JTA standard, UMJDT concepts and AspectJ, and Chapter 5 discusses the lower-level design, implementation and general architecture of TransJ, along with some fundamental concepts.

The second contribution of this dissertation is a preliminary demonstration of the feasibility, expressiveness, viability, utility, and effectiveness of *TransJ* as a framework-independent design for weaving an advice into transactions (high-level runtime abstractions) by conducting a preliminary empirical study of the impact *TransJ* on application development performance and reuse, creating a toolkit of reusable aspects of common transaction-related crosscutting concerns, and creating a suite of non-trivial sample applications that use *TransJ* (see Chapter 6). Analysis and review applications are carried out to examine the *TransJ* vs. *AspectJ* approach with regard to software performance and reuse design quality.

As a consequence of this dissertation, the developer community will have a better tool for encapsulating non-trivial crosscutting concerns into more reusable, readable, and less complex modules, rather than radically redefining them or casting them in unfamiliar usage. Thus, *TransJ* can increase software development efficiency, so the transaction system is developed faster than its AspectJ equivalent.

The preliminary evidence from this research show that the use of *TransJ* appears improve software reuse without sacrificing performance over existing AOP and OOP approaches, when the system involves or will eventually involve transaction-related crosscutting concerns. The impact of *TransJ* on software-development performance and design quality relative to reuse is confirmed by evaluating certain desirable characteristics and attributes against a quality model for transaction-related applications. To achieve this goal, we defined an extension to existing quality models with new quality factors, internal attributes, and metrics relevant to the transaction-related applications (see Chapter 7). Also, we theorized that developers should see reuse improvements while preserving the software performance relative to eight hypotheses (see Chapter 8) defined by the quality model. These hypotheses present preliminary results that lead us to believe that further experimentation with *TransJ* and refinement of its framework could prove to be very beneficial to a wide range of software systems. Chapter 9 discusses our experiment methodology, including formal approval from the Institutional Review Board (IRB) [25]; selection of the sample software applications; identification of interesting crosscutting concerns that gave us good coverage; and supporting activities, such as recruitment and training of the developers. After the experiment, we collected data from the code, journals, questionnaires, and surveys. The results of the study are presented in Chapter 10: we concluded that *TransJ*'s applications were less coupling (scattered), less complex, and required less effort and time for making code reusable across transaction-related applications. Also, they were more cohesive and oblivious without degrading the efficiency. Chapter 12 presents a conclusion, summarizing the main results of this work and indicating several directions for future research.

In summary, this dissertation presents a new framework, *TransJ*, for extending the AspectJ language to be a more powerful support for weaving at run-time. It consists of a set of base aspects and transaction-related classes—that provide well-defined reusable functionalities, which are encapsulated in reusable aspects. These reusable aspects can be configured, defined and composed in different ways to design customized application-level aspects.

In accomplishing these goals, this dissertation provides the following specific contributions:

- A *Unified Model for Joinpoints in Distributed Transactions* (UMJDT) that is a reference model to describe transaction-related joinpoints and context information that make the most sense for DTPS's. It is considered as a foundation model to extend AspectJ joinpoint model in order to support transaction aspects.
- As a technical contribution, a design and implementation of *TransJ*, including an implementation of UMJDT and JTA model that facilitate the development of the representative concepts of DTPSs, and provide the ability to track context information and weave advice into program execution before, after, or around transaction-related concepts, complete transaction, or transaction operations.
- A toolkit consisting of reusable transaction aspects of common transaction-related crosscutting concerns, which verifies the correctness of UMJDT design. These aspects include performance measuring, logging, exception handling, audit trails, data sharing, and tracing.
- A demonstration of the feasibility and utility of *TransJ* and a reusable aspect library through the implementation of DTAs and transaction aspects for those applications.
- An extension to a quality model to measure the effectiveness of *TransJ* in comparison with AspectJ. We provided an enhanced version of the current *Extended Quality Model* (EQM) [30] and *Comparison Quality Metrics* (CQMs) [54][55] that measures reusability and performance in aspect-oriented programs.

- A preliminary experiment to test hypotheses that provided evidence of improvement in code reuse without sacrificing performance when a system involved transaction-related crosscutting concerns.

CHAPTER 2

BACKGROUND

The proposed research includes and combines ideas from two different domains: transactions and *Aspect-Oriented programming* (AOP). Sections 2.1 and 2.2 present the relevant transaction concepts and models. Section 2.3 discusses the specific technologies for using transactions in Java, namely the *Transaction Application API* (JTA) and Arjuna. Section 2.3 presents key concepts of AOP necessary to understanding the contributions of this dissertation. Section 2.5 provides set of a transaction-related crosscutting concerns and discusses the role of the AspectJ for encapsulating transaction requirements.

2.1 Transaction Concepts

As mentioned, one of the objectives of this research is to implement the foundation for weaving crosscutting concerns into transactions in DTPS's. This requires identifying the logical places, i.e., joinpoints, in transaction execution where a developer might want to weave advice, as well as the kinds of information that should be available in joinpoint context.

There are many different DTPS's in use today and they vary in terms of features and implementations. However, they share commonalities in their underlying concepts of transaction distribution, management, execution, and concurrency control. We will focus on these concepts to lay a foundation for identifying transaction joinpoints and context.

As with transactions in centralized systems, a distributed transaction is a sequence of operations on the shared resources that observes the ACID properties [16][38]. The difference is that the operations of a distributed transaction execute on more than one host machine, which opens up the possibility of subsequences of those operations executing concurrently, without shared memory to help with concurrency controls.

A nested distributed transaction can be thought of as a tree of operations, instead of strict sequence. To visualize this, consider an example of a transaction-based manufacturing system that builds *Widgets* from *Goo* and *Gadgets* from *Widgets*. See Figure 2-1. The *Goo, Widget*, and *Gadgets* are all stored in "piles". The

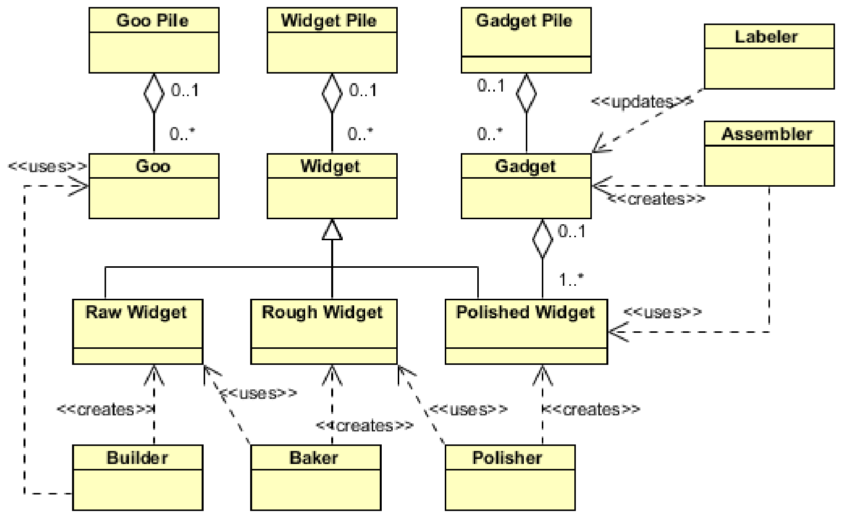


Figure 2-1. Resources in a Widget and Gadget Manufacturing System

individual objects and the piles of objects are all shared resources. This system also includes processing components that handle the manufacturing, namely, *Builders* that create *Raw Widgets* from *Goo*, *Bakers* that turn *Raw Widgets* into *Rough Widgets* and *Polishers* that refine *Rough Widgets* into *Polished Widgets*. Finally, there are *Assemblers* that create *Gadgets* from *Widgets* and *Labelers* that tag the *Gadgets* with serial

a) Transaction T1
Op1.1: Get Goo from Goo Pile
Op1.2: Give Goo to a Builder and get back a Raw Widget
Op1.3: Give Raw Widget to a Baker and get a Rough Widget
Op1.4: Give Rough Widget to a Polisher and get a Polished Widget
Op1.5: Put Polish Widget in a Widget Pile

b) Transaction T2
Op2.1: Get Widget (W1) from Widget Pile 1
Op2.2: Get Widget (W2) from Widget Pile 2
Op2.3: Give W1 and W2 to Assembler and get a Gadget, G
Op2.4: Put Gadget G in a Gadget Pile
Op2.5: Have Labeler put a tag on G

Figure 2-2. Two Sample Transactions for Constructing Widgets and Gadgets

numbers. Figure 2-2 lists two simple transactions that represent a) the construction of a *Polished Widget* and b) the construction of a *Gadget* from two *Widgets.*

Now assume that piles of *Goo*, *Widgets*, and *Gadgets* are distributed across many locations (hosts) and that *Builders* are at the same location as *Goo Piles*; *Bakers* and *Polishers* are at the same location as *Widget Piles*; and *Assemblers* and *Labelers* are close to *Gadget Piles*, but not necessarily at the same location. With this distribution of resources, transaction *T2* could execute in a distributed manner by having *Op2.1* execute in a sub-transaction, *ST2.1*, *Op2.2* execute in another sub-transaction, *ST2.2*, both on the same host as the desired *Widget Pile*, and *Op2.3-Op2.5* in a sub-transaction, *ST2.3*, on the same host as the desired *Gadget Pile*. Figure 2-3 represents this distributed transaction as a simple tree with *T2* as the root and the operations as the leaves.

T1 and *T2* are just two concrete transactions, but this system could have hundreds of similar transactions running at the same time. As in all DTPS, each transaction receives a unique identity, i.e., *Transaction Identifier* (TID), when it starts. All references to a transaction will be via this identifier. Typically, in a DTPS, a *Transaction Manager* (TM), is responsible for assigning TID's and keeping track of parent/sub-transactions relationships.

Beside TID assignment, TM's are also typically responsible for starting transactions (and sub-transactions), and ending transactions by either committing or aborting the results. A TM may also oversee the execution of transaction operations on resources and any necessary concurrency controls, such as locking,

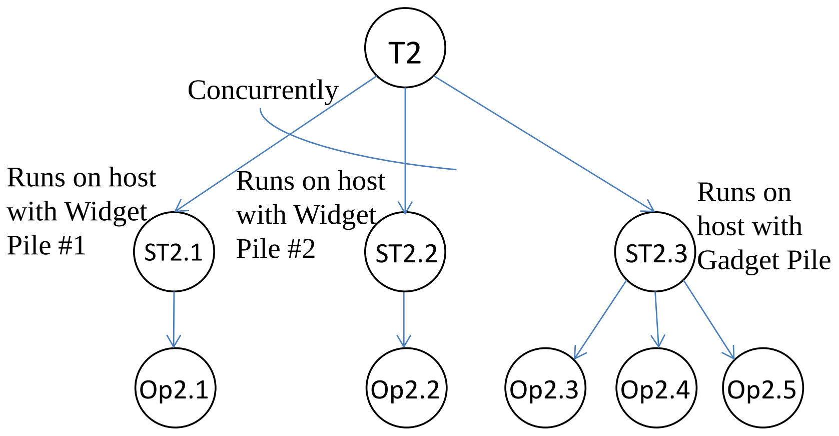

T2
Concurrently
Runs on host with Widget Pile #1
Runs on host with Widget Pile #2
Runs on host with Gadget Pile
ST2.1
ST2.2
ST2.3
Op2.1
Op2.2
Op2.3
Op2.4
Op2.5

Figure 2-3. Possible Distribution of Transaction T2

for those resources. Some DTPS delegate these responsibilities to separate components such as *Resource Managers* and *Lock Managers*, but such architectural differences are not important here. For the purpose of exploring possible transaction-related joinpoints and context information, it is important to just recognize that operation execution and concurrency control take place with respect to individual resources.

Finally, a TM can also track information about its execution environment, including information about threads of execution, processes, host machines, secondary storage, and even network connections. It may do this for a variety of reasons, including performance management, audit trails, and recovery in case of failure.

A transaction is typically broken up into two basic phases: an execution phase and a commit phase [4][16][18]. The execution phase is considered tentative, because the changes are not made permanent until the commit phase. During the execution phase, the TM performs the operations in the body within its

```
Transaction T2 = new Transaction
T2.Begin
  T2.InParallel(
     { SubTransaction T2_1 = T2. CreateSubTransaction
       T2_1.Begin
         Lock(Widget Pile 1 for Write)
         T2_1.Execute(Get Widget W1 from Widget Pile 1)
         Unlock(Widget Pile 1)
       T2_1.Commit },
     { SubTransaction T2_2 = T2. CreateSubTransaction
       T2_2.Begin
         Lock(Widget Pile 2 for Write)
         T2_2.Execute(Get Widget W2 from Widget Pile 2)
         Unlock(Widget Pile 2)
       T2_2.Commit } )
  SubTransaction T2_3 = T2. CreateSubTransaction
  T2_3.Begin
    Lock(W1 for Read)
    Lock(W2 for Read)
    T2_3.Execute(Give W1 and W2 to Assembler and get back Gadget G)
    - - Note, this operation doesn't change W1 or W2
    Lock(G for Write)
    T2_3. Execute(Put Gadget G in Gadget Pile)
    Lock(Gadget Pile for Write)
    T2_3. Execute(Have Labeler put a tag on G)
    Unlock(W1, W2, G) Gadget Pile)
  T2_3.Commit
T2.Commit
```

Figure 2-4. Pseudo Code for Distributed Version of T2

own context. Logically, the operations may result in the tentative changes to shared resources. In a commit phase, the TM will either finalize all of the tentative changes or abort the transaction.

Three common approaches to concurrency controls are *optimistic*, *timestamp-based,* and *pessimistic*. *Optimistic* approaches to concurrency control allow conflicts to occur during the tentative phases of concurrent transactions, then leave it up to the TM to detect conflicts and abort one or more transactions when they occur, using either *forward or backward validation* [27]. *Timestamp-based* approaches guarantee *serial equivalence* [42][43] by imposing an ordering on the execution of the operations in the tentative phase. *Pessimistic* approaches use locks to prevent conflicts from occurring in the tentative phase of execution. They do this by delaying operation execution or by triggering an abort (in the case of deadlock [43]). Locking schemes vary, but are all based on the premise that a transaction must hold a particular kind of lock before performing an operation.

A common and simple locking scheme consists of two types of locks: one for read operations and one write for operations [43]. The pseudo-code in Figure 2-4 includes requests for the appropriate read and writes locks, following this simple scheme.

A transaction's context information includes those pieces of data and metadata that the transaction needs to be self-contained, guarantee the ACID properties, and support correct execution of both the tentative and commit phases of execution, as shown in Figure 2-5 and Figure 2-6. Supporting correct execution of the commit phase means that the context needs to include sufficient information for the TM to decide whether

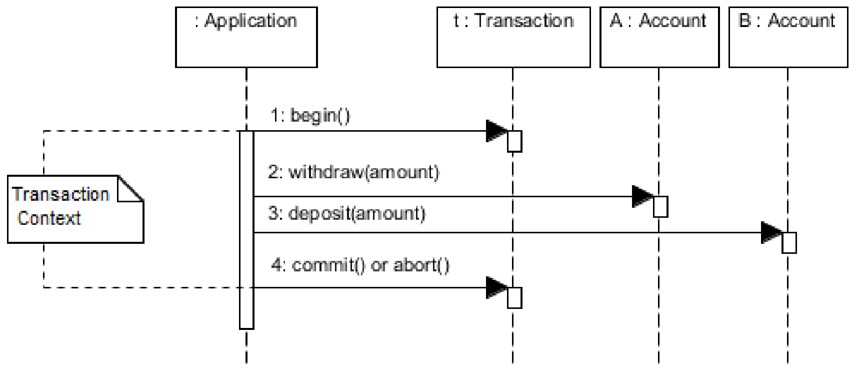


Figure 2-5. Example of Context in a Flat Transaction

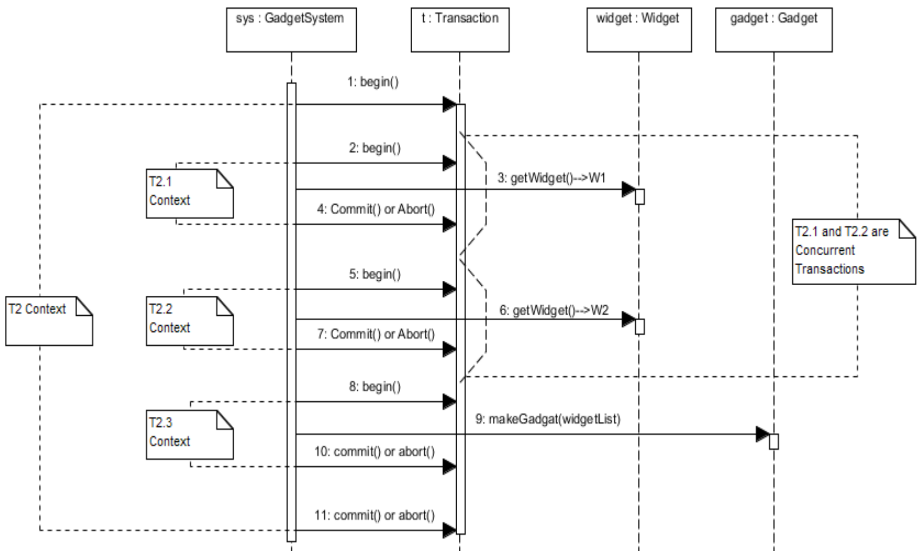


Figure 2-6. Example of Contexts in a Nested Transaction

the transaction conflicts with other concurrent transactions. However, the details of this context data depend heavily on the implementation of the DTPS, the types of concurrency control in use, and the commit algorithm. The only data that are common to virtually all DTPS are the TID and a reference (direct or indirect) to the responsible TM. Beyond these two items, a transaction's context may include many different kinds of implementation specific data, e.g., sets of tentative values, rollback logs, snapshots, lock information, timestamps, and other kinds of metadata. Therefore, any system that aims to support aspects for transaction must allow for context information to contain data that specific to a DTPS's implementation.

2.2 Overview of Java Transaction API (JTA)

The *Java Transaction Application Programming Interface* (JTA) allows applications to perform distributed transactions, that is, transactions span multiple data sources. It lets a transactional application to demarcate transaction boundaries, and allows distributed transactions to be done across multiple X/Open XA resources in a Java environment [20][40].

JTA is a specification developed under the *Java Community Process* (JCP) as JSR 907 [19]. It defines the high-level APIs between a *Transaction Manager* (TM) and the participants involved in a DTPS, i.e., transactional application (client and server applications) and the Resource Manager (RM) [19][23][24].

One of the critical advantages of using the JTA API is the separation of concerns in the DTPS, such as separating transaction management concern from the connection management concern.

In general, a distributed transaction involves many software components: *Transaction Manager* (TM), *Application Server*, *Application Program*, and *Resource Manager* (RM). Each of these players contributes to the DTPS by implementing different sets of transactional APIs and functionalities. The TM provides the management functions required to support transaction demarcation, transactional resource management, transaction completion, concurrency, synchronization, and transaction context propagation. The Application Server provides the infrastructure required to support the application run-time environment, which includes transaction-state management, such as EJB and JBoss Application Server [20][21][23][24]. The RM provides the application access to resources and supports transaction context propagation. It participates in distributed transactions by implementing a transaction resource interface used by the TM to communicate transaction association, transaction completion, and recovery [40].

Interfaces provided by transaction standards are too low-level for most DTA developers. Sun Microsystems has specified higher-level interfaces to assist in the development of distributed transactional applications: a high-level Application transaction demarcation interface, a high-level TM interface intended for Application Server, and a standard Java mapping of the X/Open XA protocol intended for transactional RM. However, these interfaces are still low-level, and require, the programmer to be concerned with state management and concurrency for transactional applications [21]. All of these interfaces occur within the *javax.transaction* package.

Figure 2-7 illustrates some of the essential JTA’s transaction concepts in perspective of our work, as described below [22][47]:

- Client Application: defines transactions and access resources within transaction boundaries.
- UserTransaction: provides applications with the ability to explicitly control transaction boundaries. It offers methods to begin, commit, rollback, and obtain the status of the transaction associated with current thread.
- Transaction: provides a control of a sequence of tasks of that are associated with target object that involves resources such as databases, which may include user interface, data retrieval, and

communications. It offers methods to begin a transaction, enlist the transactional resources, delist transactional resources, commit and rollback results for consistent state.

- TransactionManager: provides an application server with the ability to control transaction boundaries on behalf application being managed. It offers functions to maintain and encapsulate the transaction context and associate it with the calling thread.
- XAResource: defines the contract between a RM and a TM in DTPS environment, which supports association of a transaction to a resource such as Mysql database. Uses the start method to associate the transaction with the resource, and the end method to disassociate the transaction from the resource.

The RM is responsible for associating the transaction with all work performed on its data between the start and end invocations. At commit phase, the TM informs the transactional RMs to prepare (i.e., waiting for final outcome), commit, or rollback the transaction according to the *two-phase commit* protocol (2PC) [38].

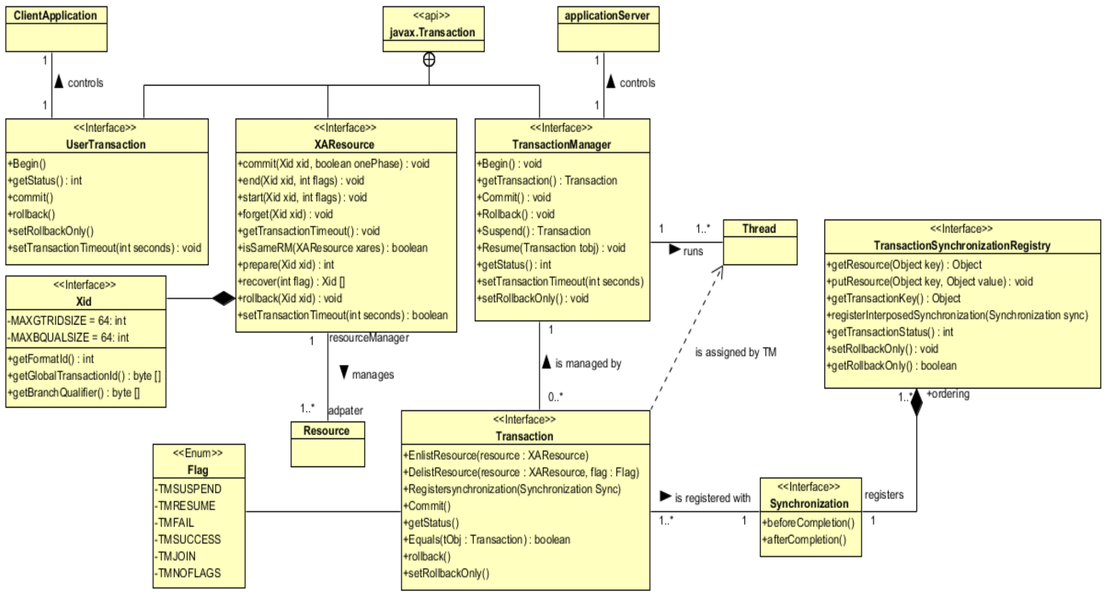

ClientApplication
<<api>>
javax.Transaction
applicationServer
1
controls
1
1
controls
1
<<Interface>>
UserTransaction
+Begin()
+getStatus() : int
+commit()
+rollback()
+setRollbackOnly()
+setTransactionTimeout(int seconds) : void
<<Interface>>
XAResource
+commit(Xid xid, boolean onePhase) : void
+end(Xid xid, int flags) : void
+start(Xid xid, int flags) : void
+forget(Xid xid) : void
+getTransactionTimeout() : void
+isSameRM(XAResource xares) : boolean
+prepare(Xid xid) : int
+recover(int flag) : Xid []
+rollback(Xid xid) : void
+setTransactionTimeout(int seconds) : boolean
<<Interface>>
TransactionManager
+Begin() : void
+getTransaction() : Transaction
+Commit() : void
+Rollback() : void
+Suspend() : Transaction
+Resume(Transaction tobj) : void
+getStatus() : int
+setTransactionTimeout(int seconds)
+setRollbackOnly() : void
1
1..*
runs
Thread
<<Interface>>
TransactionSynchronizationRegistry
+getResource(Object key) : Object
+putResource(Object key, Object value) : void
+getTransactionKey() : Object
+registerInterposedSynchronization(Synchronization sync)
+getTransactionStatus() : int
+setRollbackOnly() : void
+getRollbackOnly() : boolean
<<Interface>>
Xid
-MAXGTRIDSIZE = 64: int
-MAXBQUALSIZE = 64: int
+getFormatId() : int
+getGlobalTransactionId() : byte []
+getBranchQualifier() : byte []
1
resourceManager
manages
1..*
adpater
Resource
1
is managed by
0..*
is assigned by TM
1..*
+ordering
<<Enum>>
Flag
-TMSUSPEND
-TMRESUME
-TMFAIL
-TMSUCCESS
-TMJOIN
-TMNOFLAGS
<<Interface>>
Transaction
+EnlistResource(resource : XAResource)
+DelistResource(resource : XAResource, flag : Flag)
+Registersynchronization(Synchronization Sync)
+Commit()
+getStatus()
+Equals(tObj : Transaction) : boolean
+rollback()
+setRollbackOnly()
is registered with
1..*
1
<<Interface>>
Synchronization
+beforeCompletion()
+afterCompletion()
registers
1

Figure 2-7. JTA’s Transaction Elements

The commit process in a DTPS that involves multiple distributed resources is somewhat complex because of the transaction operations may crosscut two or more unrelated software systems and the changes to the resources must all be committed or rolled back at the same time. Moreover, creating sub-transaction within an existing transaction (parent transaction), i.e., a nested transaction model, increases the complexity of implementing the commit process due of sharing of updates from parent transaction to sub-transaction and difficulty of terminations of nested transactions within interweaving contexts which requires a committed sub-transaction’s effects become part of the parent, and become permanent only after the parent transaction commits.

JTA supports interleaving transaction contexts among multiple execution units using the same resource, as long as lock and release are invoked properly for each transaction context. However, the TM/RM cannot lock a resource for a transaction until the previous transaction has released [20]. In other words, each time the resource is used with a different transaction, the method release must be invoked for the previous transaction that was associated with the resource, and the lock must be invoked for the current transaction context.

2.2.1 *Overview of JTA Implementation*

JTA is not a product in itself, but rather a set of Java interface definitions. It’s the actual implementation of the JTA interfaces that make it possible for a DTPS to execute distributed transactions. Accordingly, a vendor-specific JTA implementations referred to as a TM’s or transaction services are needed to actually use the functionality defined in these interfaces. The Implementation details can vary with respect to JTA concepts, but in general their capabilities are similar. For example, Narayana (Arjuna) [47], JBossTS [20][21][22], Atomikos [23], and Bitronix [24] are widely popular open source JTA implementations that touch the architectural concepts of a DTPS. Arjuna, JbossTS, Bitronix, and Atomikos platforms have been used as open-source TMs that implement the interfaces of JTA, which map the industry standard, X/Open XA Interface, to Java. One implementation may provide more flexibility over another in handling a particular situation, but these differences only impact the implementation of the ideas in this research and not the core contributions. In this dissertation, we focus on the Arjuna. Arjuna allows the JTA to support nested distributed transactions.

One of the cornerstones of the *Java Enterprise* (J2EE) platform implementations is the *Enterprise JavaBean* (EJB) component. This component provides transactional support implementation of the basic JTA functionalities as a part of its platform. The J2EE specification [44][45] describes the *Java Transaction Service* (JTS), how JTA adapts J2EE, and tasks that a certified application server must cope. The implementation of JTS included in J2EE supports distributed JTA transactions. This is required to access different data sources within a distributed transaction control. This may allow a J2EE container to weave the functionality encapsulated in aspects written in an AOPL, like AspectJ. We used this platform to build a sample of applications (see Appendix A.)

JbossTS is the default TM for *Jboss application server* (JbossAS). JbossJTA is a middleware solution that integrates easily with other Jboss frameworks, such as WildFly Application Server, or directly with standalone Java programs, such as *Enterprise Application Platform* 6.3 (EAP 6.3). All of the Narayana JTA classes and interfaces occur within the *com.arjuna.ats.jta* package (see Section 11.1.1 for more details). In the implementation of experiment applications, we used the JbossJTA as the TM for local and distributed JTA applications.

2.3 Overview of Aspect-Oriented Programming

AOP is an extension to OOP that allows developers to extract and untangle secondary concerns from the primary features of an application. It is difficult to define what constitutes a secondary concern in general because it depends on the purpose of the software being built. However, secondary concerns often show up in less-than-expertly-designed object-oriented software as similar snippets of code scattered across multiple modules or tangled into methods that primarily serve other purposes. A common example is tracing or logging in a data processing application, where the developers want a chronology of the execution for system verification, audit-trail, or performance-monitoring measurement reasons. To do this, they might insert logic throughout the code that writes various messages or statistics to a file. Eventually, these log-writing code snippets become scattered across the software and tangled in otherwise cohesive methods.

An AOPL, like AspectJ [33], allows a developer to remove all of the log-writing code from the main application and place that logic in an aspect, which is a class-like abstract data type. An aspect can include data members, methods, nested types and everything else a class can include. However, they can also include advices and pointcuts. An advice is like a method because it implements some specific behavior; however, it is not invoked like a method. Instead, the AOPL's compiler or runtime environment weaves the advice into the system so it is executed at specific places and time defined by pointcuts. A pointcut is a pattern that identifies a set of joinpoints, which are best characterized as intervals within the program's execution flow. Examples of joinpoints in typical AOPL's include the execution of a method or the setting of a property. Consequently, their start and end points map to specific elements of the code, called shadows, which correspond to places where those intervals may start or end.

When advice executes, it can access context information about the joinpoint at which it was invoked. This context includes the location of the joinpoint (i.e., the shadow) and runtime information about the objects involved. Some of the context information is static and therefore can be computed during weaving; other context is dynamic and depends on the objects involved in the joinpoint.

2.3.1 *AOPL's Implementation: Tools and Frameworks*

There are two main approaches for implementing the logic of crosscutting concerns (i.e., aspects) of an application: asymmetric and symmetric. An *asymmetric* approach depend on the notion that there is a core body of code that is then augmented with aspects (i.e., weaving *aspect-base*) [83][90]. Consequently, the aspects are typically implemented in an AOP-extension (i.e. AspectJ) of the base language. On the other hand, in a *symmetric* approach, there is no distinction between an aspect and a base code (i.e., composition *base-base*) [60][83].

Implementation of the crosscutting concerns requires the understanding of how these concerns interact with other parts of the application. In other words, the *JoinPoint Model* (JPM) defines the locations in a code program where/when crosscutting concerns can be applied, a way to choice these locations and a means of affecting the behavior at these locations.

The weaving of advice into the shadows is an automated process, wherein several of AOP tools require a special language processor, called an *Aspect Weaver*. It performs weaving that allows to coordinate the co-composition of the aspects and application modules. The current most used tools are AspectC++ [34], AspectJ [8][33], AspectWorkz [9], Jboss AOP [10][20], Spring AOP [11][35]. These tools are built on principles, which are Advice, Aspect, Joinpoint, and Pointcut. However, they are different in executions and syntaxes, static or dynamic analysis, binding approach, expressiveness, advice weaving-approaches (i.e., compile time, runtime time or load time), and their overall academic and industry enhancements.

Jboss Application Server provides an AOP facility [10]. This facility represents the one of the most aggressive effort to support aspects in an enterprise environment to date. It uses joinpoint constructs called interceptors that describe points in the application code that are to be aspected. The AspectWorkz as Jboss AOP, defines aspects and advice code blocks in regular Java classes, while pointcut and advice are defined using XML configuration files. They perform weaving at runtime using a custom class loader. The JBoss class loader performs the aspect weaving at deployment time.

Spring AOP is a proxy-based AOP, thus performs weaving at runtime through a dynamic proxy. It supports only method execution joinpoints - unlike AspectJ, where it supports all joinpoints, such as fields, methods, constructors, catch blocks in exception handling, etc.

The AJDT plug-in for Eclipse offers powerful support for modular design and implementation of real world quality software, which includes platform based tool support for AOSD with AspectJ [3][4][27][36][37]. AspectJ is considered the de facto standard and the most widely used AOP framework for modeling crosscutting concerns due to its Java-like structure is derived from the JPL, powerful expressiveness, and debugging abilities, even though it has a little overhead in terms of memory usage and time. Moreover, the AspectJ weaver can insert an advice block from more than one aspect into application base code. When the weaver matches a pointcut signature with a joinpoint defined in two or more different aspects, it may weave the advice blocks in any order with unpredictable results. In this research, we would limit our implementation and experience to AspectJ for defining the transaction-related crosscutting concerns.

2.4 Crosscutting Concerns in Transaction

Even though the distribution [36][106], concurrency [43], persistency [36][49], and failures [15][49] are major concerns in many current large-scale DTAs, current research show that AOP methodology is very powerful and using it through AspectJ can enable development of concise, modular, efficient, flexible and cost-effective source code in shorter span of time [7][14][30][31][32][107][110][111]. Improvement in modularity, decrease in complexity and enhancement to the reusability of the software are also reported. Such an improvement often proposes mechanisms to disclose the code and structure of the scattered and tangled snippets in the execution of the applications.

The number and variety of crosscutting concerns in a DTPS are perhaps infinite. However, for illustrative purposes, we will consider just one here. Imagine that we would like to optimize the Gadget manufacturing system such that Widgets were created just in time, by making sure there are always some Widgets in a pile, but never in excess.

Such flow-control or timing issues could be considered a secondary crosscutting concern to the basic Gadget assembly problem. By talking with the domain experts, we would probably discover a couple of basic rules that govern when the Widget product needs to be speed up or slowed down. An object-oriented programmer could embed the logic for these rules into the implement of the *Builder, Baker, Polisher,* or some other set of components. With some skill, it is possible that the object-oriented programmer might even be able to do this in a modular and reusable way.

With transaction aspects, an AOP programmer, however, would have a much similar option. Basically, the programmer would encapsulate the logic for speeding up or slowing down widget production into an aspect, maybe called something like *WidgetProductionSpeedControl*. This aspect would include advice that could be woven before (or around) any operation that accesses a Widget pile. The advice's logic would speed up Widget product if the pile was getting too small or slow it down if the pile was getting too large. The aspect would also include a simple pointcut that defined a pattern for all relevant joinpoints. The original application code would not need to be aware of the new production-speed control logic. In fact, because of this obliviousness, it could be tested with or without the speed control functionality without any reprogramming of the system.

AspectJ by itself does not help programmers to encapsulate such logic in modular, reusable abstractions. To make things worse, this new production-speed control logic might change (evolve) as other features evolve in the system such as performance. What makes adding this new production-speed control logic difficult is its crosscutting nature: the implementation of production-speed control, using the object-oriented modularization technique, requires change in various places in the code. In other words, the production-speed control's code cannot be encapsulated in one place and is scattered throughout the code. The programmers need a modular reasoning approach to discover the code and the structure of the crosscutting concerns that access the Widget pile; whereas she would most likely need global reasoning when using traditional OO techniques [29]. However, it is not clear what discipline would help programmers in AspectJ obtain modular reasoning for such concerns. Specifically, the obliviousness property of AspectJ conflicts with the ability to reason about programs in a modular fashion. Therefore, the developer may end up struggling with unnecessary coupling (i.e., lack of obliviousness), compromised flexibility, and less reusable code. Briefly, the application developer must be aware that the separation does not imply a common semantic decoupling.

To resolve this situation, the programmer needs a mechanism that not only does allow expressing the transaction crosscutting features without breaking the existing system, but also helps to do it in a modularized and reusable fashion. One way of accommodating this through providing a mechanism to leverage low-level aspects, i.e., base aspects, to provide customizable run-time trace transaction, designed to enhance understanding of transaction systems infrastructure software. Each of the low-level aspects is individually reusable and does know about the specific details in transactions' contexts in which it is used. This makes it possible to share runtime information among the aspects that allow the programmer to encapsulate the code for a particular concern in one place and modularize reusable crosscutting functionality within aspects. Each of these aspects provides a well-defined common reusable functionality, which, as it

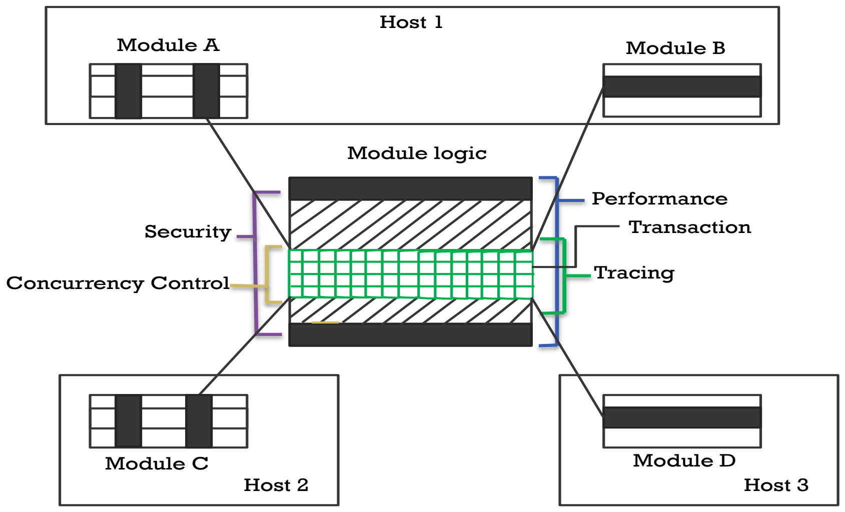


Figure 2-8. Transaction Code Tangling in a Module Logic and Code Scattering of Transaction Concerns

turns out, is often needed in applications to perform a specific behviour. To allow the aspects to be used in other transaction-related contexts, they should be carefully configured, designed and composed to be reusable.

Figure 2-8 shows how a module may look like in a system in which transaction crosscutting concerns are not modularized. The module is implementing a transactional feature in the module logic, however, there are pieces of code that belong to other concerns in the system, such as concurrency control, performance, security, audit trail, and tracing. In other words, the module implements pieces of multiple transaction concerns and the transaction itself. This presence of portions of implementation of multiple concerns in a module is referred to as code tangling. The implementation of a crosscutting concern like concurrency control can be present in many other modules, even in other crosscutting modules spread in many other distributed hosts.

Code scattering can occur in two different ways [76]. It occurs when a piece of implementation repeatedly appears in many modules, as shown in Figure 2-8. It can also be the result of complementary fragments of implementation appearing in several modules that may be distributed over many hosts as listed below. For instance, concurrency controls, like lock and unlock operations, occur in different modules in a system, even for the same resources. In a nutshell, Code tangling and code scattering negatively affect software design and development by causing poor traceability, lower productivity, lower code reuse, poor performance, poor quality, harder evolution, and may lower efficiency [26].

In fact, a DTPS could be further complicated by implementing other transaction-related crosscutting concerns as listed below, because they pertain to multiple parts of a core or base transaction system.

- Transaction Monitoring.
- Measuring transaction system throughput.
- Measuring transaction-based application responsiveness between clients and distributed servers, in other words, measuring turnaround performance for an individual transaction, the nested distributed transactions, and the complete execution of a transaction.
- Optimization using distribution whose implementation is distributed across multiple DTPS's modules to optimize the throughput of the DTPS workload-based on the most likely behavior of a

transaction. It helps programmers to determine the most efficient way to execute a transaction by considering the possible behaviors on shared resources.

- Implementing a data-sharing optimization mechanism to share of context information across hosts only when necessary.
- Simulating transactions in a DTPS to test a specific process.
- Tracking consistent and secure transactions or handling authentication permissions in transaction domain.
- Adding a new security-level to validate state of shared data coming to/from data sources.
- Measuring the amount of threads being used to serve a specific transaction.
- Logging is a typical crosscutting concern that span multiple objects or modules in different level of abstractions within a DTPS. It records information about a transaction's operation executions and distributed resources in a specific developer-defined format.
- Implementing an audit trail mechanism can be complicated and not modular in a DTPS, because its code is spread over multiple units and transaction components in order to capture a historical record to find out who has access, and what transaction he/she has performed during a given period of time, and when the data is changed.
- Implementing *Just-In-Time* (JIT) strategy that would be a crosscutting concern since it touch more parts of the transaction contexts to gather and deliver the context information only when necessary.
- The notification in a DTPS that is often scattered and tangled because it uses to detect critical errors, exceptions, time-based expiration, and the invalid state.
- Implementing security, specifically authorization and validation rules can adversely affect DTPS's implementation due to increase the complexity of existing business logic to track consistent, secure transactions, and handle authentication permissions in DTPSs.
- Scattering the logic of concurrency and synchronization techniques throughout different elements of the DTPS and implementing similar snippets of code in such elements.
- The nature of the volume, and velocity of transactions reinforce the overall complexity of a DTPS.
    - Volume of transactions: the amount of the transactions (committed and aborted) in an application to deal with the transaction system requirements.

- Velocity of transactions: a rate of transactions that are executed per minute.
- Number of transactions that are involved in the deadlock.

2.4.1 *Role of the AspectJ for Encapsulating Transaction Requirements*

Although AOP promises increase readability, reusability and maintainability, it also requires special attention from the developers [51][52]. Building an AO application is a complicated process that involves a lot of effort. Also, since AO software development is fairly a new approach, it lacks a certain methodology that can be applied. In our experiment, we use AspectJ as a representative AOPL and use transaction concepts as a fundamental paradigm to handle transaction-related crosscutting concerns. For example, there are various thoughts on what AspectJ can achieve and what it cannot. In [28], Kienzle provides a good discussion that analyzes the limitations of AOP, in which the author try to use AOP techniques to separate transaction-related concepts from the core other parts of the application. However, this attempt shows that transaction must be kept in mind throughout the entire application development, that it cannot be completely extracted to a separate aspect. This supports the argument that AspectJ by itself is insufficient to encapsulating transaction-related crosscutting concerns.

For example, if a programmer wants to measure turnaround performance for transactions in AspectJ, the programmer would have to implement some advices to start a process that would capture the time at which a transaction is begun and other advice they would capture the time at which the transaction is committed/aborted and then compare the two times (see Chapter 6). However, begin and commit/abort logic for the transaction may be in separate modules, may be separated in the execution flow by an undetermined amount of time, and may even be handled on separate execution threads or separate hosts. Furthermore, the nested transaction may start many sub-transactions at the same time, and the advice would have to weave the time of a begin sub-transaction with the commit time of the sub-transaction. In a nutshell, the weakness of AspectJ is that its pointcut designators and joinpoint contexts are limited to standard programming constructs, and do not deal the high-level run-time abstractions, i.e., entire transaction.

AspectJ, like other AOPLs, does not allow a programmer to define the independent aspect to this concern, because it cannot capture behavior that spans across several modules and expose the related context information that's associated with the complete execution of a transaction. AspectJ's pointcut designators remain weak relative to weaving crosscutting concerns into transaction abstractions, which we can’t easily define by available designators. Specifically, programmers cannot deal with code transaction as predicates, which select transaction joinpoints based on their properties and the relevant context. However, we believe that AspectJ has a good potential to deal with code constructs, so programmers would only be able to weave concerns into the underlying transaction operations, such as begin, commit, abort, start, and end. Also, the programmers would have to explicitly code mechanisms for tracking transaction contexts (transaction context, lock context, and operation context).

To address this problem for transactions, we will develop an extension to AspectJ framework, called *TransJ*, that allows developers to define pointcuts in terms of transaction abstractions and that automatically keeps track of context information for transactions. The following section provides a high-level overview of *TransJ*’s architecture.

CHAPTER 3

TRANSJ ARCHITECTURE

3.1 Overview

Figure 3-1 represents the relationships between the principal parts and functions in *TransJ*. In other words, it is an architectural block diagram of *TransJ*, in which the colored blocks represent relevant conceptual layers, and arrows depict dependencies among these layers. It describes the *TransJ*'s design at a higher level, with less detailed description aimed more at understanding the overall of its concepts and less in understanding the details of implementation [56][114]. We adopt a strategy of top-down to the design of the *TransJ* with a layered architectural design [56][57], in which each layer embodies a reusable function or the logical component and provides services to the layer above it and uses the services of the layer below it.

Overall *TransJ* represents a set of principles that provide an abstract framework to promote code reuse through managing the complexity of DTPS's design while preserving the performance. Specifically, the core *TransJ* infrastructure layer enables aspect-oriented developers to treat transactions as first-class

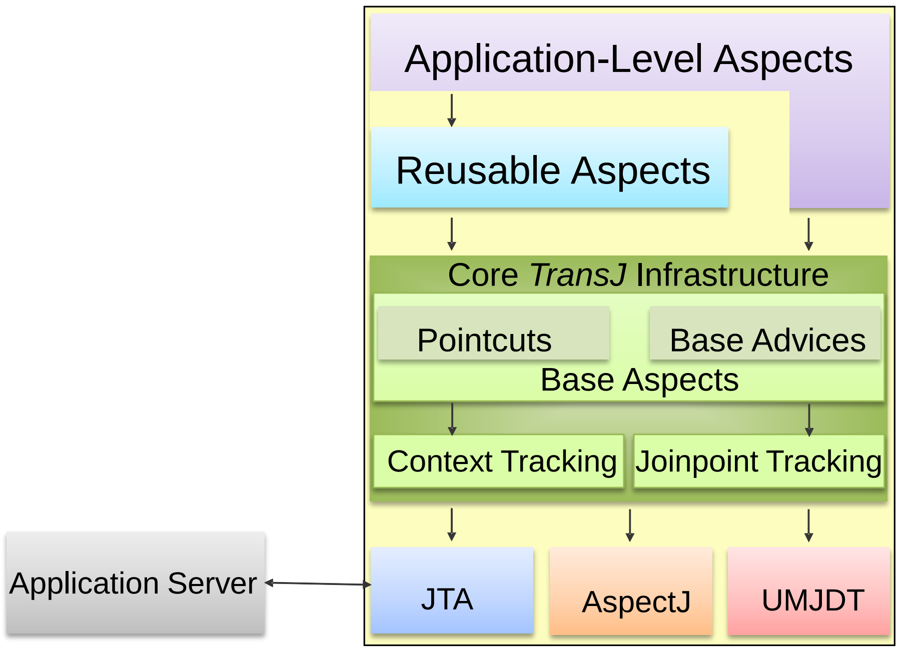


Figure 3-1. The Architectural Pattern to *TransJ*

concepts into which AspectJ framework can weave crosscutting concerns in a modular way, i.e., transaction aspects. This promotes greater enhancements, obliviousness, localization along with code reusability, but may have some performance impacts as well. The following sub-sections provide some necessary details about each of the layers in the order from top to bottom, that will ultimately set the stage to assess whether the benefits are achieved.

3.2 Application-level Aspect Layer

Application layer represents an abstraction layer that contains a set of common transaction-related aspects, which encapsulate base-application requirements. The aspects of this layer are aspects of aspects. In other words, we can build application-level aspects either by extending the abstract aspects provided by the reusable aspects or/and base aspects in core *TransJ*. This type of aspects can help developers to encapsulate the business requirements of the application into high-level aspects. For example, a manufacturing system would have a number of application-level aspects, such as production speed control, measuring performance, logging, load balancing and more. Among them, the logic of Widget production speed control concern can be written at the application-level using the base aspects, which extends pointcut constructs that select appropriate operation joinpoints, where *WidgetProductionSpeedControl* aspect should be woven (see Section 6.4).

Application-level aspects can use directly either the base aspects or the abstractions provided by *TransJ* to access metadata that related to transactions, operations and the affected resources that are pulled by context trackers. This type of information can describe the expected behavior of the transaction with respect to the context in which it is running such as lock context, operation context, or transaction context (Section 4.1.2 for more details). Our goal is for resultant application-level aspects are easy-to-code, more reusable, understandable, predictable, flexible and modular than similar concerns, programmed in AspectJ or OOP fashion. (Chapter 6 provides more details)

3.3 Reusable Aspect Layer

The definition of the reusable layer within the scope of *TransJ* is a layer responsible for providing a set of helpful aspects that encapsulate common transaction-related crosscutting concerns and exposing relevant context data that application aspects must consider once weaving advices. Overall the reusable aspects represent general crosscutting concerns commonly found in applications with significant transaction requirements, and therefore can be woven in DTPSs where a transaction-related concern is applicable. In other words, this layer represents a toolkit-like collection of transaction aspects that developers should find useful for in several of DTAs. These reusable aspects depend on a set of the core *TransJ* aspects that can decrease the development time to program application-level aspects, and make them more understandable, reusable, predictable, and oblivious. To ensure that is done effectively, we need appropriate, precise specifications of such aspects that can then be used to understand the behavior of the DTPSs and introduce behaviors into complex like nested distributed transactions. The core *TransJ* provides specifications for reusable aspects. This kind of aspect is conceptually inspired from the key transaction joinpoints defined in the UMJDT. Chapter 6 provides more details and examples.

3.4 Core *TransJ* Infrastructure Layer

The *TransJ* is a library that introduces a transaction JPM on top of AspectJ JPM. It consists of components for tracking transaction contexts and joinpoints; base aspects that core transaction abstractions; and a collection of pointcuts for transaction operations. The base aspects include base advices that embody and augment the behavior of a transaction; and a collection of pointcuts for gathering context information that can be used in the advice code.

We specify the behavior of transaction aspects in terms abstraction concepts that to any DTPS built using JTA and XA [20]. These abstract aspects define one or more pointcuts and items of advices that will execute when transaction reaches joinpoints matching these pointcuts.

The *Context and Joinpoint tracking* (*trackers*) encapsulate hooks into the underlying transactions subsystems, such as JTA transaction and UMJDT transaction, in which pull relevant context information for transaction base aspects and keep track the start and end points of the joinpoint. If those changes, one only needs to replace or extend these trackers. The base aspects make use of the context information provided by the context tracking and allow reusable or application-level aspects specific to individual transactions. The joinpoints defined in the *TransJ* core Infrastructure give the reusable aspect and application-level aspect convenience, reusable pointcuts for transactional joinpoints.

3.5 Unified Model for Joinpoints in Distributed Transaction (UMJDT)

UMJDT is a formal description of common knowledge related to transactions. It describes a common conceptual understanding about transactions to encapsulate any complex relationship, which can exist in a DTPS. Specifically, it unifies DTPS concepts related to a) transactions in general, b) the kinds of information that comprise their context, and c) events that represent interesting time points/places for when/where the crosscutting concerns might augment an application's core functional or the underlying transaction processing system. Therefore, it has become a foundation for weaving transaction aspects into high-level abstraction, i.e., transaction (see Chapter 4 for more details.)

3.6 Java Transaction API (JTA)

JTA is another foundation part of *TransJ* architecture. It offers a procedural interface to transactions and resources, including several methods that allows an application developer to start, join, commit, and abort transactions (See Chapter 2 for more operations and details). Begin operation, which starts a new transaction or a nested transaction within an already ongoing one; commit operation, which attempts to commit the current transaction; abort operation, which forces the transaction to rollback, and more. In addition, it provides multithreaded transaction models provide additional operations to allow threads to join an ongoing transaction (join operation), which allows the calling thread to join the transaction with the current transaction context. *TransJ's* pointcuts tied to the JTA constructs.

3.7 AspectJ Framework

The *TransJ* infrastructure realizes the UMJDT [18] for AspectJ [8]. In AspectJ, the joinpoints are certain well-defined points in the execution flow of a Java program. These include method and constructor calls or executions, field accesses, object and class initialization, and others. Pointcut designators allow an aspect developer to choose a certain set of joinpoints, which can further be composed with Boolean operations to build up other pointcuts. It is also possible to use wild cards when specifying, for instance, a method signature [8]. The transaction pointcuts in core *TransJ* infrastructure will build on standard AspectJ pointcut designators. In addition, the core *TransJ* infrastructure would do not constrain to use any AspectJ feature, such as programmer-defined pointcuts, advice, inter-type declarations, etc. (See Section 2.4.1 for more details.)

3.8 Application Server

Typically, to handle a transaction across multiple distributed resources, we need an application server. It is not part of the architecture of the *TransJ*, but it enables distributed transactions across multiple hosts. For our implementation of TransJ and experience, we have selected the Red Hat Jboss *Enterprise Application Platform* (EAP) 6.4. This platform provides all transaction-related concepts, implements JTA API, concurrency control (locking) and supports nested distributed transaction systems [21].

3.9 Initial Theoretical Comparison of *TransJ* to AspectJ

The layers represented in this Chapter can offer software system developers with a number of important benefits when it comes to managing the complexity of transactions in applications.

We chose to implement TransJ in AspectJ since it is currently one of the most popular, expressiveness and stable AOPLs. Nevertheless, the current AspectJ capabilities are not sufficient to express crosscutting concerns related to transactions, we believe the *TransJ* can do better to localize and encapsulate crosscutting concerns that apply to transactions as execution units since it provides:

- *A Design Space for New Language Constructs for Transactions: Better Abstractions for Transactions*
    - *TransJ* aspectizes transaction concepts to provide loosely coupled abstractions, which allow programmers directly weave advices into high-level abstractions, like transactions. Therefore, *TransJ* provides better abstractions that unify transaction concepts. In comparison, AspectJ weakly encapsulates and modularizes transaction concerns due to limited abstractions of the underlying AspectJ and need multiple pointcut definitions to overcome different types of transaction abstractions for an individual or a nested distributed transaction such as the concurrency control, and transaction itself.
- *Improved the Reusability:*
    - *TransJ* allows programmers to capture transaction-related concerns (e.g., security, performance measurements, exception handling), into well-modularized entities (e.g., reusable aspects and

application-level aspects) that would be woven into high-level runtime abstractions, in which developers can implement these concerns in cohesive and loosely coupled aspects. In comparison, AspectJ needs to program intricate forms of coupling, (i.e., tight coupling), which in turn might jeopardize reusability.

- *Joinpoint Model Formalizes Transactional Joinpoints*:
    - In *TransJ*, programmers can define pointcuts using terms are related directly to general transaction concepts, which can access the transaction state and the context information about the joinpoint at which it was invoked. In comparison, AspectJ provides no vocabulary for defining transaction-related joinpoints and pointcuts.
- *High-level Encapsulations, and Localized the Design Decisions:*
    - *TransJ* provides a rich set of reusable aspects that correspond to crosscutting concerns that occur in many DTPSs. These aspects can be extended from abstract base aspects to implement transaction-related crosscutting concerns in several applications. The reusable aspects also localize internal design decisions, and encapsulate many complex mechanisms such as tracking the context information, and joinpoints. With AspectJ, the developer would need to define complex data structure and explicit mechanisms to combine context information with the effect of the advice code to arrive at the richer behavior of the transaction method.
- *Improved Modularity and Obliviousness*:
    - *TransJ* completely separates transactional interfaces (begin, commit, rollback, setlock, release, etc.) from the main functional transaction concepts, and have these encapsulated within code invoked through specific aspects. Therefore, developers can capture the crosscutting concern for transactions in terms of general begin, commit, abort, start and end joinpoints, regardless of the underlying concepts of transaction, distribution, management, execution, synchronization and concurrency control. With AspectJ, application programmers hardly write understandable aspect code for transactions, because programming abstractions vary with the underlying transaction mechanisms or characteristics. For example, the transactions might span many different threads or hosts and be interleaved with the execution of many other concurrent transactions.
- *Better Ways to Detangle Transaction Constructs from Core Application:*

  - AspectJ doesn't provide any abstractions to help alleviate redundant and tangled code of the distributed transaction requirements. In comparison, *TransJ* is a layer of abstraction on top of AspectJ and JTA helps to program the transaction aspects in a uniform manner, hence, it makes them more detangled, reusable and flexible aspects than similar crosscutting concerns that are programmed in AspectJ's fashion. In addition, *TransJ* approach provides the logic source code is much easier to read and understand, without the clutter of the code needed to support transaction-related crosscutting concerns.

- *More Fluid to Code Transaction-Related Crosscutting Concerns:*
  - AspectJ directly cannot provide pointcuts to gather transaction contexts that may be shared among multiple threads or hosts. In doing so, the developers would need to define considerably more complex pointcuts to program the logic of transaction concerns away from the underlying application logic. On the contrary, it becomes very easy to program transaction concerns using *TransJ*'s pointcuts with fewer lines of code. The core *TransJ* infrastructure layer of abstraction on top of AspectJ helps developers to track contexts and joinpoints, which make context information more fluid than similar crosscutting concerns, programmed in directly AspectJ.
- *Conceptual Model Captures Transaction Context Information*
  - *TransJ* provides low-level distributed aspects that track context information in order to perform the expected weaving into abstractions that span multiple threads of execution and may be interleaved with concurrent execution of similar abstraction. With AspectJ only, developers would need complex data structures and explicit mechanisms in order to pull together all of the relevant data that needs to make up a transaction context to weave an applicable advice.
- *Better Organized Transaction Concerns:*
  - In *TransJ*, application-level aspect shows the code of transaction concerns more classy, structured, and predictable than the same concerns programmed in AspectJ.

The results of the experiment in this dissertation (see Chapter 10) provide some preliminary evidence that *TransJ* truly realized these benefits.

CHAPTER 4

REFERENCE MODEL FOR TRANSJ

This Chapter describes a conceptual model that provides a theoretical foundation for *TransJ*, namely jointpoints, contexts and tracking.

4.1 High-level Overview

Overall *TransJ* enables the separating of complex transaction concerns into manageable cohesive concepts and promotes greater reuse while maintaining the efficiency.

4.1.1 *Transaction Events and Possible Joinpoints*

Figure 4-1 uses colored dots in the timeline of the transaction to represent a formal semantics for joinpoints along with a specified time interval. These dots actually correspond to the high-level abstractions that help in managing transaction complexity and enabling compositional logic. These are dubbed transactional joinpoint events. These events allow the simple solutions to the design of interesting high-level abstractions in transaction programming. For example, managed concurrency can be treated as a high-level

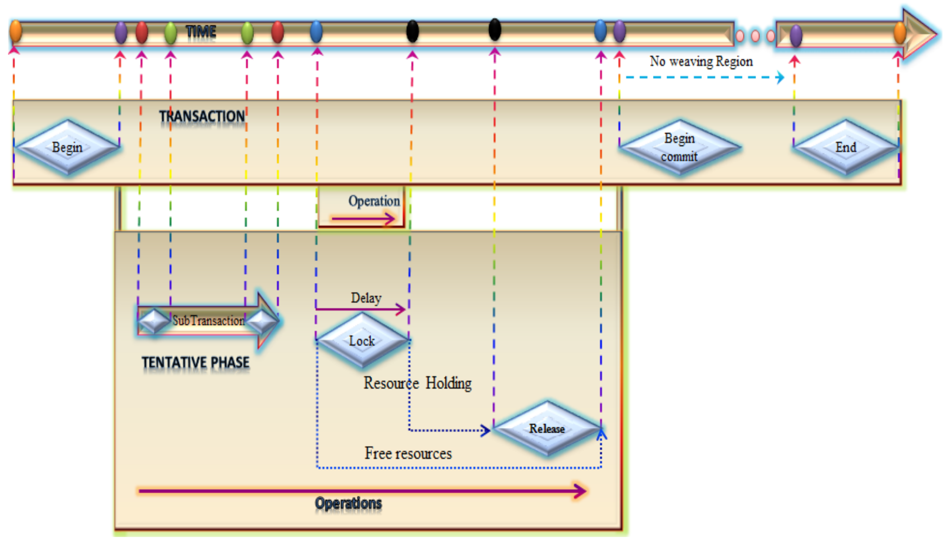
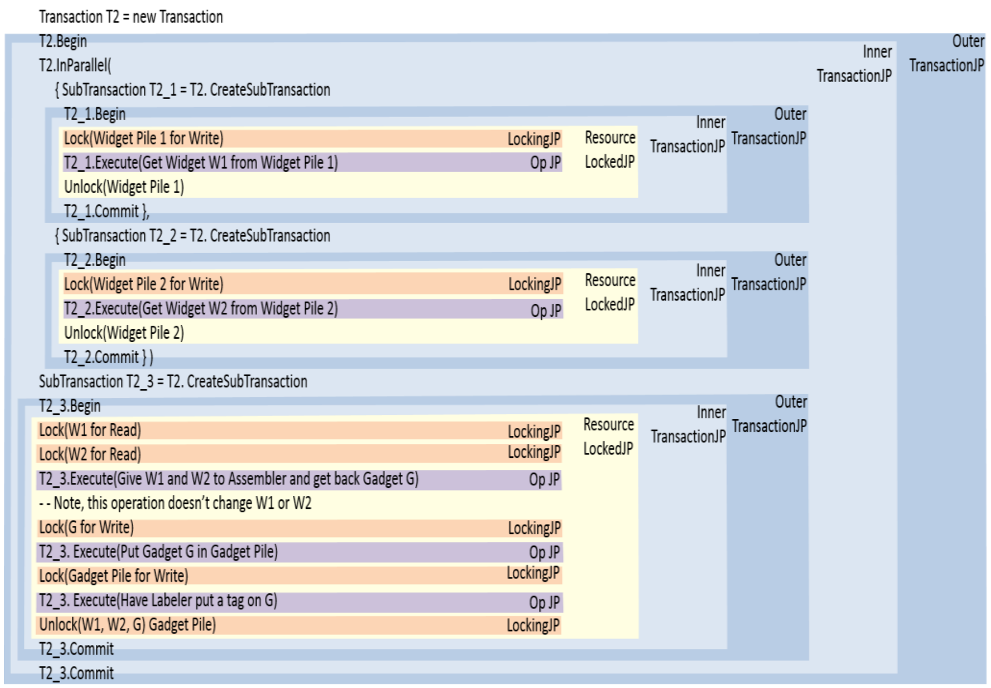


Figure 4-1. Transaction Events and Possible Joinpoints

abstraction to lock and release different resources, whereas in other a non-abstract, non-modular concept is required. Likewise, the entire transaction can also be implemented as an abstract transactional event, which introduces a new type of joinpoint constructs. As described below, Figure 4-1 represents different kinds of potential joinpoints in transactions as follows: an outer region that maps the interval that spans the entire transaction execution, starting before the transaction event begins and ending after the event is committed or aborted. These transaction events are termed *Begin Event* and (*Commit Event* or *Abort Event),* respectively; an inner region that maps the interval that spans the entire transaction execution, and a sequence of intervals (e.g., sub-transactions or operations), starting after the transaction event begins and ending before beginning the event is committed or aborted. These transaction events are termed *Begin Event* and (*Commit Event or Abort Event),* respectively; a marked-acquired lock region that maps to the interval when the lock is requested, starting before the transaction invokes a lock event for a specific resource and ending after the lock event is granted or denied. These events are termed *Begin Request lock Event* and *End Request lock Event,* respectively; a resource holding region that maps to the interval when the lock is held, staring after the transaction granted the required lock and ending before the transaction mark the lock as free. These transaction events are termed *hold Event* and *release hold Event,* respectively; Operation (arrow) represents a transaction operation region that maps to the interval when the operation is executed, staring before the transaction operation is invoked and ending after the transaction operation exits. These transaction events are termed hold Event and release hold Event, respectively.

4.1.2 ***Potential Joinpoints and the Scope of the Context***

From an advise-weaving perspective, joinpoints map to places where weaving occurs – hence the use of “point” in the name. However, from an execution perspective, a joinpoint represents a logical interval of time in a flow of execution. It has a beginning and an end, and advice can be woven into the flow of execution before, after, or around it. This section presents Figure 4-2 as a pseudo-code for the implementation of T2 annotations that illustrates five new types of joinpoints for DTPS’s: outer transaction, inner transaction, resource locked, locking, and operation. Each type of joinpoint is shown in a different color. This section also discusses interesting metadata that advice might want to use, and therefore should be part of joinpoint contexts.



```
Transaction T2 = new Transaction
T2.Begin
T2.InParallel(
  { SubTransaction T2_1 = T2. CreateSubTransaction
    T2_1.Begin
    Lock(Widget Pile 1 for Write)                                  LockingJP
    T2_1.Execute(Get Widget W1 from Widget Pile 1)                 Op JP
    Unlock(Widget Pile 1)
    T2_1.Commit },
  { SubTransaction T2_2 = T2. CreateSubTransaction
    T2_2.Begin
    Lock(Widget Pile 2 for Write)                                  LockingJP
    T2_2.Execute(Get Widget W2 from Widget Pile 2)                 Op JP
    Unlock(Widget Pile 2)
    T2_2.Commit } )
SubTransaction T2_3 = T2. CreateSubTransaction
T2_3.Begin
Lock(W1 for Read)                                                  LockingJP
Lock(W2 for Read)                                                  LockingJP
T2_3.Execute(Give W1 and W2 to Assembler and get back Gadget G)    Op JP
- - Note, this operation doesn't change W1 or W2
Lock(G for Write)                                                  LockingJP
T2_3. Execute(Put Gadget G in Gadget Pile)                         Op JP
Lock(Gadget Pile for Write)                                        LockingJP
T2_3. Execute(Have Labeler put a tag on G)                         Op JP
Unlock(W1, W2, G) Gadget Pile)                                     LockingJP
T2_3.Commit
T2_3.Commit
```

Figure 4-2. Pseudo Code for Distributed Version of T2 and the Potential Transaction Joinpoints within the Scope of the T2's Context

An Outer Transaction Joinpoint represents the interval that spans the complete execution of a transaction, starting just before the tentative phase and ending after the completion of the commit phase. This kind of joinpoint would allow a programmer to introduce advice before, after or around an entire transaction. However, because it starts before the beginning of the tentative phase, any "before" advice would not have access to the target transaction's context information. However, it would have access to a parent transaction's context, which would be particularly important for advice before or around sub-transactions.

An Inner Transaction Joinpoint is similar to an Outer Transaction Joinpoint, except that it starts just after the tentative phase begins and ends just before the commit phase ends. Advice woven before this kind of joinpoint would have access to the target transaction's context.

A Resource-locked Joinpoint represents the interval that spans the time when a lock is held, starting after acquiring the lock and ending just before its release. Advice woven before, after or around this type of joinpoint would have access to metadata about the lock, the associated resources and, of course, the transaction.

A Locking Joinpoint represents the interval that spans a lock request. In other words, it begins as a request is made and ends when the request is granted or denied. Advice woven before, after, or around a Locking Joinpoint can access metadata about the type of lock being requested or the resource.

An Operation Joinpoint is an interval that spans one operation in the execution of the tentative phase of a transaction. Such advice would access metadata about the operation and the affected resources, as well as the transaction at large.

4.1.3 *Unified Conceptual Model for Joinpoints in Distributed Transactions (UMJDT)*

Figure 4-3 shows part of the UML model, called the *Unified Model for Joinpoints in Distributed Transactions* (UMJDT), which captures the key ideas for the new transaction joinpoints and related context information [18]. The class labeled *TransJP* is a generalization of the joinpoints as discussed above. By definition, each is associated with a StartEvent, but may not have an EndEvent if the interval is still in process. Every TransJP can also reference a context that holds all the relevant statics and runtime information for the joinpoint. Aspect advice will use this context to access a wide variety of information such as operations in progress, resources, and current execution environments.

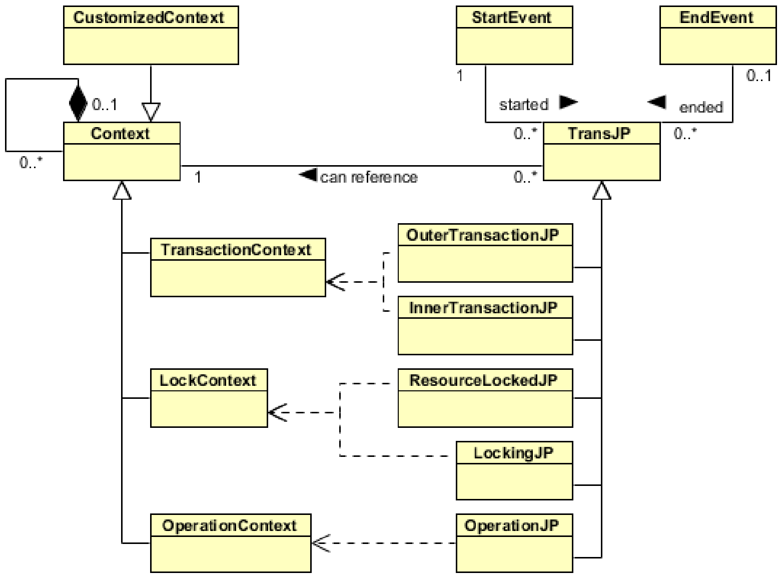


Figure 4-3. Part of the Unified Model for Joinpoints in Distributed Transactions

However, there are three special kinds of contexts, and the actually kind of context that a TransJP directly accesses depends on the TransJP specialization. For example, a LockingJP directly accesses a LockContext.

Contexts can be composited into a hierarchy of objects, as indicated by the recursive aggregation relationship connected to the Context class. Although Figure 4-3 does not show all the possibilities and constraints, a LockContext can be part of a TransactionContext, which could in turn be part of another TransactionContext (i.e., for a parent transaction.)

Contexts may also be extensible or customizable objects. In other words, the base system that makes transaction aspect possible will provide classes for Context and its three immediate specializations. It also projects hooks for extending those classes, either through specialization, plugs-in, or even other kinds of aspects, so programmers can use context details that are specific to a particular DTPS or DTPS-based applications.

CHAPTER 5

IMPLEMENTATION OF A TRANSJ TOOL SET

This chapter discusses the implementation of the *TransJ*. Sections 5.2 through 5.6 provides the technical information about the *TransJ*, including the transaction concepts that integrate well with the AspectJ features found in modern programming languages. Figures 5-1 shows a UML paradigm that presents low-level aspects, high-level joinpoints, high-level contexts, and trackers. The following sections provide the details.

5.1 Overview

The motivation for building such an abstract independent framework is the observation that transaction concerns cannot be completely separated at a higher level, and the implementation level as well. That means, there may be common base logic between these concerns. One example of a conflict would be that most concurrency control approaches do not work with an in-place update. At the same time, being able to distinguish read from write or update transaction operations is a common functionality that is required by transaction.

Motivated by inability to extract separation of concerns completely, the abstract framework for transactions applies AO design techniques to separate transaction concepts, and specifies a set of small well-defined aspects, each providing a specific cohesive sub-functionality. The following sections describe the aspects that are the end results of the separation and encapsulation processes. As a result, the application developer can make use of transactions as he/she pleases, and does not have to worry about possible interference problems, i.e., coupling.

5.2 Transaction Joinpoints

As mentioned earlier, joinpoints represent places and times where/when advice can be executed. In AspectJ, they correspond to constructors, methods, attributes, and exceptions. In *TransJ*, they correspond to abstractions that may span into interleaved multi-threaded or distributed hosts. The UMJDT serves as a

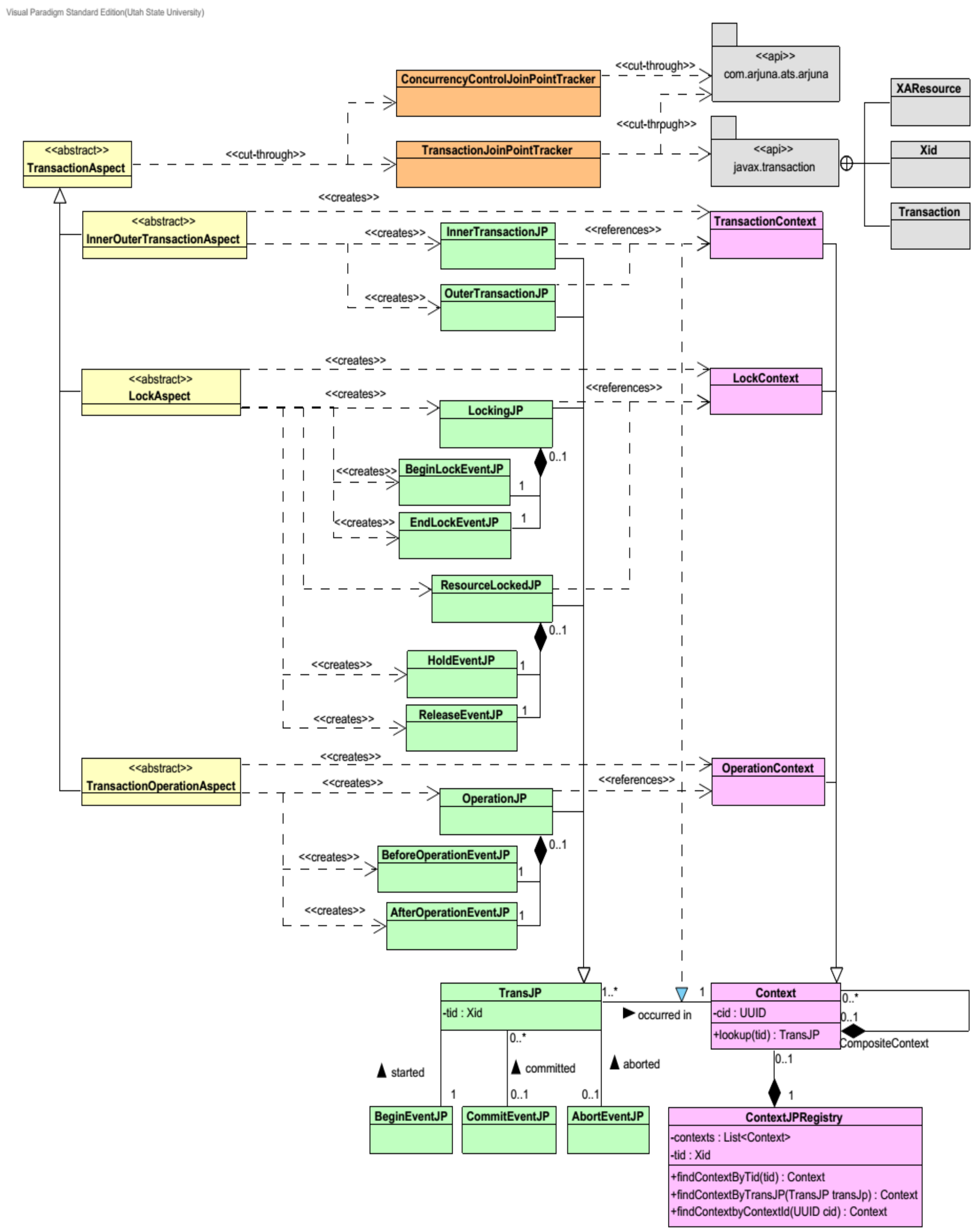


Figure 5-1. Part of the *TransJ* Event, Joinpoints, Contexts and Aspects

foundation for formalizing transaction joinpoints, which fall into three general categories: transaction joinpoints, operation joinpoints, and concurrency control joinpoints. These categories refer to three different contexts: transaction context, operation context, lock context, respectively.

Figure 5-1 presents a general joinpoint that is labeled *TransJP* that encompasses the logical connection between transaction-event joinpoints. In other words, it is designed to carry out generic transaction joinpoints, such as creating transaction-event joinpoints and finding where a specific transaction is involved. Each event can be associated with many other events, with at most one thread. One transaction can have multiple threads, and a host can process multiple transactions concurrently. For example, in a distributed nested transaction system, a transaction *T1* can begin executing on the thread *Th#1* which corresponds to a begin event, and then allows the transaction to commit or abort for some other thread *Th#2*.

The green boxes in Figure 5-1 are *TransJ* classes that implement joinpoints for different kinds of contexts. Such joinpoints offer a natural abstraction in term of events, enable the explicit definition of complex crosscuts by means of event pattern, and accommodate very general behaviors of a transaction.

Overall TransJP represents a joinpoint for the entire execution transaction, as well as joinpoints for a sequence of sub-transactions within a transaction scope, for a sequence of operations within an operation scope, and for lock/release concurrency operations within a lock scope. TransJP defines three event types: begin event, commit event, and abort event. The begin event is when something happens at a particular point, i.e., begin point, related to the setting up of transaction flow control. The commit or abort event is when something happens at a particular point, i.e., commit or abort point, related to the end of the transaction execution flow. These events are mapped to three event joinpoints, respectively: *BeginEventJP*, *CommitEventJP or AbortEventJP*. Each one implements a single joinpoint for an individual transaction event. *BeginEventJP* represents an execution point of the code into which advice can be woven, when TransJP related to the begin event of the transaction occurs in the transaction system. *CommitEventJP or AbortEventJP* represents an execution point of the code into which advice can be woven, when TransJP related to the commit or rollback event transaction, respectively, occurs in the transaction system. TransJP is specialized into five types of joinpoints: InnerTransactionJP, OuterTransactionJP, LockingJP, ResourceLockedJP, and OperationJP.

- *InnerTransactionJP* represents the region of code or period during which a specific transaction code is executed, where advice can be woven in, when TransJP occurs after the begin event and before the commit/abort event (prior the end of the transaction execution flow.)
- *OuterTransactionJP* represents the region of code or period during which a specific transaction code is called, where advice can be woven in, when TransJP occurs before the begin event and after the commit/abort event (after the transaction has completed.)

The UMJDT states that every commit or abort event must have a corresponding begin event. In other words, a begin event can exist without a commit or abort event, but not conversely. The events of these kinds of joinpoints are capable of keeping track of transactions that occur in multiple threads within distributed transactions.

Inner/Outer transaction joinpoints have direct access to the target transaction's context, where the woven advices occur before, after or around these joinpoints. They refer to a transaction context concept, i.e., TransactionContext, which contains the relevant transaction information that is delivered at execution time to a proper transaction knowledge, as shown in Figure 5-1.

- *LockingJP* represents a joinpoint for acquiring the resources used to perform a particular transaction operation. In other words, it represents the region of code or the period during which a specific transaction code region is executed, where advice can be woven in, starting when a begin lock request event is sent to the RM/lock manager and ending when the lock request event is granted or refused. The beginning and end of the lock request code are associated with two events that are capable of keeping track of the lock request within the lock context, as shown in Figure 5-1.
    - *BeginLockEventJP* represents an execution point of the code, where advice can be woven into, when *LockingJP* runs before executing a set lock event for acquiring the specified resource within the lock context associated with the target transaction.
    - *EndLockEventJP* represents an execution point of the code, where advice can be woven in, when *LockingJP* runs in place of a set lock event to get the lock that has been granted or refused for the specified resource within the lock context associated with the target transaction.

- *ResourceLockedJP* represents a joinpoint for complete a lock is held. In other words, it represents the region of code or the period during which a specific transaction code region is executed, where advice can be woven in, when a hold event occurs after setting the lock and end before releasing the lock. The demarcation points of the resource locked joinpoint correspond to two events that are capable of tracking the status of locked resources associated with a target transaction within the lock context, as shown in Figure 5-1.
    - *HoldEventJP* represents an execution point of the code, where advice can be woven in, when a *ResourceLockedJP* occurs the after executing a set lock event to hold the specified resource within the lock context associated with the target transaction.
    - *ReleaseEventJP* represents an execution point of the code, where advice can be woven in, when a *ResourceLockedJP* occurs before executing a release event to unlock the specified resource within the lock context associated with the target transaction.
- *OperationJP* represents a joinpoint for complete a transaction operation. In other words, it represents the region of code or period during which a specific transaction operation code region is executed, where advice can be woven in, when a transaction occurs invoking any transaction method, (i.e., a method is annotated with a transactional annotation) from within the scope of a transaction, which indicates whether a method will be executed within an operation context associated with the target transaction. This joinpoint contains *BeforeOperationEventJP* and *AfterOperationEventJP* for keeping track of the status of all transaction operations.
    - *BeforeOperationEventJP* represents an execution point of the code, where advice can be woven in, when an *OperationJP* occurs before executing a reflective access to the information about a transaction operation associated with the target transaction.
    - *AfterOperationEventJP* represents an execution point of the code, where advice can be woven in, when an *OperationJP* occurs after executing the reflective access to the state available and information about a transaction operation within the operational context associated with the target transaction.

Each TransJP refers to a specific context that contains all the relevant statics and runtime information for its joinpoint/s: TransactionContext, OperationContext, or LockContext.

5.3 Contexts for Joinpoints

Advice can be executed before, after, or around various *contexts*. *TransJ* adds transactions to the list of possible contexts, but unlike the contexts in AspectJ, a transaction is not tied to a single programming construct; rather, it consists of loosely-coupled abstractions. *TransJ's* contexts include various pieces of interesting data and metadata that woven advice might use, e.g., identifier, status, sets of tentative values, rollback logs, snapshots, lock information, timestamps, and other kinds of metadata.

Figure 5-1 presents the composite pattern for the context. The context represents a base class for context primitive, whereas the context represents a composite class that maintains a collection of subcontexts in term of a tree structure to represent part-whole hierarchies: *Transaction Context*, *Lock Context*, and *Operation Context.* These contexts represent concrete primitive contexts; for example a lock context or an operation context can be part of a transaction context, which in turn can be part of the parent transaction context.

- *TransactionContext* encapsulates the transaction information that has to be shared among all the outer and inner transaction joinpoints, such as transaction identifier, starting time, commit time, abort time, sub-transactions, status, timestamp, tentative values for resources, etc.
- *LockContext* encapsulates the lock information related to underlying resources along with their transactions, such as a locked time, a released time, time-out, status of shared resources, lock mode, lock result, lock owner, tentative values to the update resource, etc. This information has to be available to the LockingJP and ResourcelockedJP.
- *OperationContext* encapsulates information about the sequence of the transaction operations in the transaction's body, and operations in progress, etc. This information has to be available to the OperationJP.

Generally, each context includes the location of the joinpoint and runtime information about the transaction objects involved. For this, *TransJ* contexts exhibit one or more attributes associated the events that are represented joinpoints as discussed above. *TransJ* considers these joinpoint events as largely independent, while a context considers them as interrelated through its call transaction concepts that would lead to a more reusable and robust implementation. These events keep track and record the transaction identifying information, e.g., the transaction identifier (TID), for all types of joinpoints. When advice executes, it can access the context information about the joinpoint at which it was invoked. Each specialized context provides different kinds of information that should be available to weave a relevant advice. In other words, the *TransJ*'s context is dynamic and depends on the target transaction objects involved in the joinpoint.

5.4 Registry for Contexts

Transaction aspects dynamically introduce context information for all *TransJ* joinpoints. When a joinpoint event occurs, e.g., BeginEventJP, *TransJ* creates an instance of a joinpoint class, e.g., InnerTransactionJP, that further correlates it with other events in the same joinpoint associated with a target transaction, and then adds the instance of the joinpoint to a relevant context, which contains a collection of joinpoints of the target transaction, and then adds the context to the registry, which contains a collection of contexts, namely, the ContextJPRegistry, as shown in Figure 5-1.

When a joinpoint aspect, e.g., InnerOuterTransactionAspect, discovers a relevant transaction joinpoint and correlates it with other appropriate joinpoints that belong to the same transaction. Advices in a transaction-related aspect can access these joinpoint objects to obtain context information, like a transaction's start time, identifier, status, or the underlying lock information. This task can be facilitated if every context maintains a list of all the transactions that have accessed it. ContextJPRegistry provides this functionality by keeping a list of all transaction-related contexts that have interacted with the target transaction. On this, ContextJPRegistry represents a repository for all transaction-related contexts, which provide relevant information for advices associated with the target transaction at execution time.

5.5 Trackers

Depending on the location of the joinpoint event, the appropriate part of the context knowledge should be gathered in those contexts, i.e., transaction context, operation context, and lock context. Behind the scenes, *TransJ* uses context gathering mechanisms, namely joinpoint tracker aspects, that are based on context-aware transaction abstractions and employ joinpoint events to dynamically gather the relevant information.

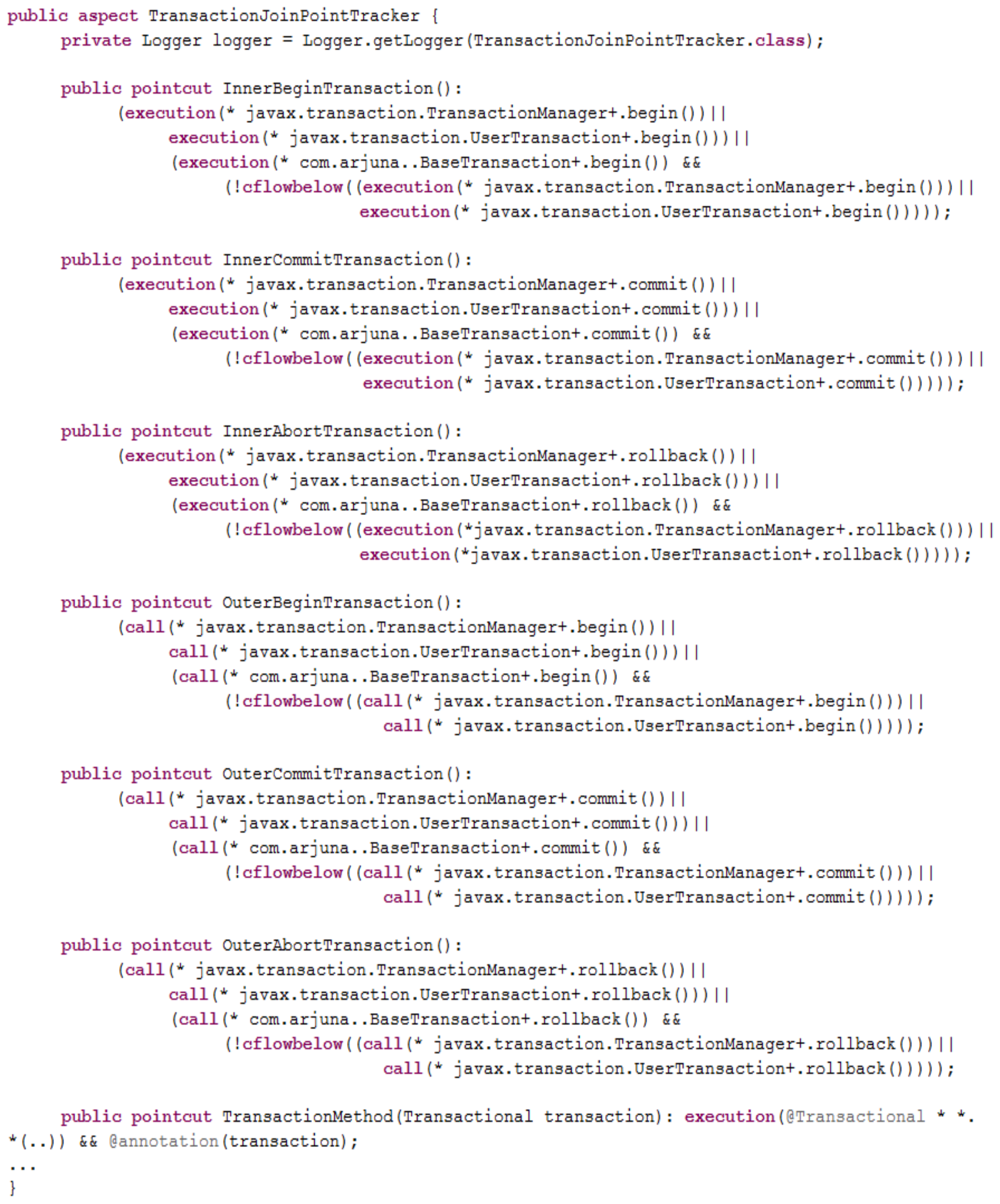
The trackers work as monitors [32][94] that perform pattern matching on transaction events, to track individual events and to organize them into high-level transaction-related contexts. Since the monitoring of transactions is itself a crosscutting concern, trackers are implemented as aspects that weave the necessary monitoring logic into places where a transaction event may take place. *TransJ* can support many different kinds of transaction joinpoint trackers, Figure 5-1 shows two special types of trackers, namely *TransactionJoinPointTracker* and *ConcurrencyControlJoinpointTracker.*

5.5.1 ***TransactionJoinPointTracker***

The transaction joinpoint tracker is an aspect that hides transaction-related abstractions in the core transaction application. It crosscuts begin, commit, abort, and transaction operation "@transactional" abstractions and defines a set of elegant and parameterized pointcuts. These provide benefits for sharing states between advices while overcoming the syntactic and semantic variations, defined on standard JTA and Arjuna pre-built libraries, i.e., javax.transaction and com.arjuna.ats.arjuna. These pointcuts are rich enough to encapsulate abstractions for transaction-related concepts of the client and server sides, e.g., UserTransaction and TransactionManager, respectively.

TransactionJoinPointTracker discovers a relevant joinpoint of the transaction based on the knowledge of access transactions, i.e., access to external transactions or access to internal transactions. Hence, *TransJ* creates seven clean, well-encapsulated transaction-related abstractions for all kinds of types begin, commit, rollback, and transactional annotation (shown in Figure 5-2), summarized as follows:

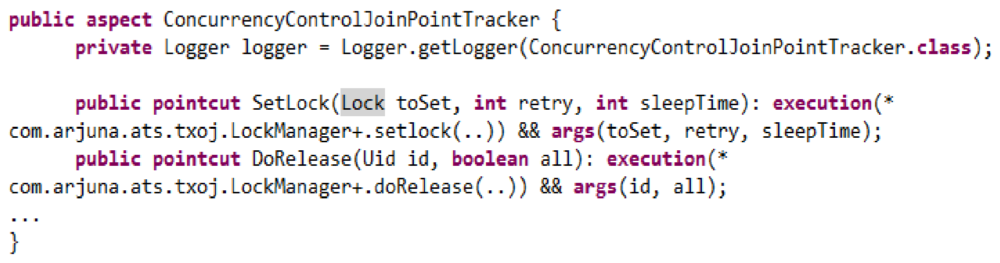
- Transaction pointcuts for begins: These pointcuts unify syntactic and semantic variations in JTA libraries, i.e., JTA API and Arjuna API, and crosscut outer and inner transaction begin abstractions.
- Transaction pointcuts for commits: These pointcuts unify syntactic and semantic variations in JTA libraries, i.e., JTA API, and Arjuna API, and crosscut outer and inner transaction commit abstractions.
- Transaction pointcuts for abort: These pointcuts unify syntactic and semantic variations in JTA libraries, i.e., JTA API, and Arjuna API, and crosscut outer and inner transaction rollback abstractions.



```
public aspect TransactionJoinPointTracker {
     private Logger logger = Logger.getLogger(TransactionJoinPointTracker.class);

     public pointcut InnerBeginTransaction():
          (execution(* javax.transaction.TransactionManager+.begin())||
               execution(* javax.transaction.UserTransaction+.begin()))||
               (execution(* com.arjuna..BaseTransaction+.begin()) &&
                    (!cflowbelow((execution(* javax.transaction.TransactionManager+.begin()))||
                                   execution(* javax.transaction.UserTransaction+.begin())))));

     public pointcut InnerCommitTransaction():
          (execution(* javax.transaction.TransactionManager+.commit())||
               execution(* javax.transaction.UserTransaction+.commit()))||
               (execution(* com.arjuna..BaseTransaction+.commit()) &&
                    (!cflowbelow((execution(* javax.transaction.TransactionManager+.commit()))||
                                   execution(* javax.transaction.UserTransaction+.commit())))));

     public pointcut InnerAbortTransaction():
          (execution(* javax.transaction.TransactionManager+.rollback())||
               execution(* javax.transaction.UserTransaction+.rollback()))||
               (execution(* com.arjuna..BaseTransaction+.rollback()) &&
                    (!cflowbelow((execution(*javax.transaction.TransactionManager+.rollback()))||
                                   execution(*javax.transaction.UserTransaction+.rollback())))));

     public pointcut OuterBeginTransaction():
          (call(* javax.transaction.TransactionManager+.begin())||
               call(* javax.transaction.UserTransaction+.begin()))||
               (call(* com.arjuna..BaseTransaction+.begin()) &&
                    (!cflowbelow((call(* javax.transaction.TransactionManager+.begin()))||
                                   call(* javax.transaction.UserTransaction+.begin())))));

     public pointcut OuterCommitTransaction():
          (call(* javax.transaction.TransactionManager+.commit())||
               call(* javax.transaction.UserTransaction+.commit()))||
               (call(* com.arjuna..BaseTransaction+.commit()) &&
                    (!cflowbelow((call(* javax.transaction.TransactionManager+.commit()))||
                                   call(* javax.transaction.UserTransaction+.commit())))));

     public pointcut OuterAbortTransaction():
          (call(* javax.transaction.TransactionManager+.rollback())||
               call(* javax.transaction.UserTransaction+.rollback()))||
               (call(* com.arjuna..BaseTransaction+.rollback()) &&
                    (!cflowbelow((call(* javax.transaction.TransactionManager+.rollback()))||
                                   call(* javax.transaction.UserTransaction+.rollback())))));

     public pointcut TransactionMethod(Transactional transaction): execution(@Transactional * *.
*(..)) && @annotation(transaction);
...
}
```

Figure 5-2. A Code Snippet of TransactionJoinPointTracker

- Transaction pointcuts for transactional annotations: This pointcuts unify syntactic and semantic variations in JTA libraries, i.e., JTA API, Arjuna API, and EJB, and crosscut transaction operation that annotated with Transactional abstractions.



```
public aspect ConcurrencyControlJoinPointTracker {
	private Logger logger = Logger.getLogger(ConcurrencyControlJoinPointTracker.class);

	public pointcut SetLock(Lock toSet, int retry, int sleepTime): execution(*
com.arjuna.ats.txoj.LockManager+.setlock(..)) && args(toSet, retry, sleepTime);
	public pointcut DoRelease(Uid id, boolean all): execution(*
com.arjuna.ats.txoj.LockManager+.doRelease(..)) && args(id, all);
...
}
```

Figure 5-3. A Code Snippet of ConcurrencyControlJoinPointTracker

5.5.2 *ConcurrencyControlJoinpointTracker*

The concurrency control joinpoint event tracker is an aspect that hides concurrency control abstractions in core transaction applications. This aspect crosscuts the syntactic and semantic variations that exist on standard JTA, e.g., pre-built Arjuna library, and unifies them into a set of parameterized pointcuts in set lock and release lock abstractions. These pointcuts are rich enough to encapsulate and manage all concurrency-related abstractions and styles related to the locking and unlocking of shared resources in distributed transactions. Hence, *TransJ* provides two clean, well-encapsulated transaction-related abstractions for setlock and doRelease, (shown in Figure 5-3). These are summarized as follows:

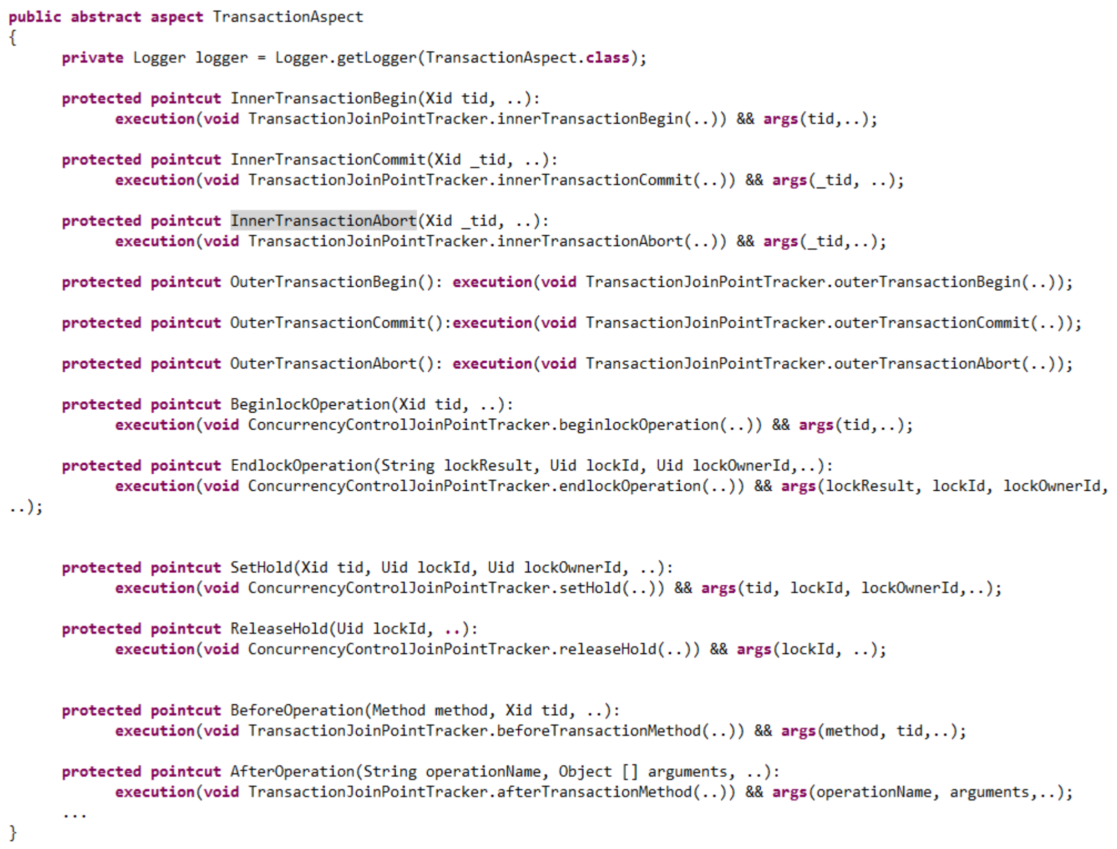
- Concurrency Control pointcut for setlock: It crosscuts setlock operation for the lock managers in Arjuna API while requesting to hold a specified resource to the associated transaction.
- Concurrency Control pointcut for doRelease: It crosscuts doRelease operation for the lock manager in Arjuna API while releasing a lock of a specified resource from the associated transaction.

5.6 Base Transaction Aspects

TransJ implements transaction-related crosscutting concerns as aspects derived from transaction aspects that cut through their respective joinpoint trackers. These aspects are derived from abstract *TransactionAspect,* which provides high-level concrete pointcuts that dynamically track different transaction abstractions, i.e., begin, commit, abort, setlock, doRelease, and transactional abstractions, as shown in Figure 5-4. In *TransJ* architecture, the core infrastructure contains three kinds of base transaction aspects, InnerOuterTransactionAspect, OperationAspect, and LockAspect, as shown in Figure 5-1.

The pointcuts in the *TransactionAspect* take a list of objects as parameters, because this is how concrete aspects based on these pointcuts can access transaction-related context information. We bind context data to pointcut variables, which can then be used to parameterize advices. This allows concrete aspects to be parameterized and configures different joinpoints, which enable reusable aspects to be customized in different contexts and thus increase the reusability of aspects (see Chapter 10 for more details). The base aspects consist of three distinct abstract aspects corresponding to three different kinds of contexts, as mentioned earlier, and extend the *TransactionAspect* with pointcut abstractions that are meaningful to those contexts (see Figure 5-4). On this, developers can create their own application-level transaction aspects that inherit these aspects and include advice based on these pointcuts.

- *InnerOuterTransactionAspect* extends the *TransactionAspect* with pointcuts for transaction beginnings and the transaction ends, as shown in Figure 5-5. It involves begin, commit and abort



```
public abstract aspect TransactionAspect
{
      private Logger logger = Logger.getLogger(TransactionAspect.class);

      protected pointcut InnerTransactionBegin(Xid tid, ..):
            execution(void TransactionJoinPointTracker.innerTransactionBegin(..)) && args(tid,..);

      protected pointcut InnerTransactionCommit(Xid _tid, ..):
            execution(void TransactionJoinPointTracker.innerTransactionCommit(..)) && args(_tid, ..);

      protected pointcut InnerTransactionAbort(Xid _tid, ..):
            execution(void TransactionJoinPointTracker.innerTransactionAbort(..)) && args(_tid,..);

      protected pointcut OuterTransactionBegin(): execution(void TransactionJoinPointTracker.outerTransactionBegin(..));

      protected pointcut OuterTransactionCommit():execution(void TransactionJoinPointTracker.outerTransactionCommit(..));

      protected pointcut OuterTransactionAbort(): execution(void TransactionJoinPointTracker.outerTransactionAbort(..));

      protected pointcut BeginlockOperation(Xid tid, ..):
            execution(void ConcurrencyControlJoinPointTracker.beginlockOperation(..)) && args(tid,..);

      protected pointcut EndlockOperation(String lockResult, Uid lockId, Uid lockOwnerId,..):
            execution(void ConcurrencyControlJoinPointTracker.endlockOperation(..)) && args(lockResult, lockId, lockOwnerId,
..);


      protected pointcut SetHold(Xid tid, Uid lockId, Uid lockOwnerId, ..):
            execution(void ConcurrencyControlJoinPointTracker.setHold(..)) && args(tid, lockId, lockOwnerId,..);

      protected pointcut ReleaseHold(Uid lockId, ..):
            execution(void ConcurrencyControlJoinPointTracker.releaseHold(..)) && args(lockId, ..);


      protected pointcut BeforeOperation(Method method, Xid tid, ..):
            execution(void TransactionJoinPointTracker.beforeTransactionMethod(..)) && args(method, tid,..);

      protected pointcut AfterOperation(String operationName, Object [] arguments, ..):
            execution(void TransactionJoinPointTracker.afterTransactionMethod(..)) && args(operationName, arguments,..);
      ...
}
```

Figure 5-4. A Code Snippet of TransactionAspect

joinpoints to demarcate the transaction scope. It defines six pointcuts: *iTransactionBegin*[2], *iTransactionCommit*, *iTransactionAbort*, *oTransactionBegin*[3], *oTransactionCommit* and *oTransactionAbort*. These pointcuts crosscut *TransactionJoinpointTracker* to establish a transaction context on the client application and the application server sides of each executed transaction. The oTransactionBegin creates an OuterTransactionJP and instantiates a transaction context. The oTransactionCommit or oTransactionAbort retrieves the matching OuterTransactionJP from the target TransactionContext in ContextJPRegistry and ends a transaction after a client or transaction manager invokes a commit or abort joinpoint event. The iTransactionBegin creates an InnerTransactionJP and starts a transaction when a client or

```
public abstract aspect InnerOuterTransactionAspect extends TransactionAspect{
     private Logger logger = Logger.getLogger(InnerOuterTransactionAspect.class);

     public pointcut oTransactionBegin(OuterTransactionJP _outerTransactionJp) :
execution(* InnerOuterTransactionAspect+.Begin(OuterTransactionJP)) &&
args(_outerTransactionJp);

     public pointcut oTransactionCommit(OuterTransactionJP _outerTransactionJp):
execution(* InnerOuterTransactionAspect+.Commit(OuterTransactionJP)) &&
args(_outerTransactionJp);

     public pointcut oTransactionAbort(OuterTransactionJP _outerTransactionJp) :
execution(* InnerOuterTransactionAspect+.Abort(OuterTransactionJP)) &&
args(_outerTransactionJp);

     public pointcut iTransactionBegin(InnerTransactionJP _innerTransactionJp) :
execution(* InnerOuterTransactionAspect+.Begin(InnerTransactionJP)) &&
args(_innerTransactionJp);

     public pointcut iTransactionCommit(InnerTransactionJP _innerTransactionJp):
execution(* InnerOuterTransactionAspect+.Commit(InnerTransactionJP)) &&
args(_innerTransactionJp);

     public pointcut iTransactionAbort(InnerTransactionJP _innerTransactionJp) :
execution(* InnerOuterTransactionAspect+.Abort(InnerTransactionJP)) &&
args(_innerTransactionJp);
     ...
}
```

Figure 5-5. Extended Parametrized-Pointcuts in InnerOuterTransactionAspect

[2] The transaction pointcut is initialized with a lowercase letter that indicates where the joinpoint is located, (i) stands for inner transaction, e.g., iTransactionBegin means inner begin transaction.

[3] The transaction pointcut begins with a lowercase letter that indicates where the joinpoint is located: e.g., (o) stands for outer transaction; oTransactionBegin means outer begin transaction.

transaction manager executes a begin event, and then retrieves the matching target TransactionContext from the ContextJPRegistry and adds the InnerTransactionJP. The iTransactionCommit or iTransactionAbort retrieves the matching InnerTransactionJP from the target TransactionContext in the ContextJPRegistry and adds the commit joinpoint event or abort joinpoint event, as shown in Figure G-1 (Appendix G). Developers can use this kind of aspect to weave advice before, after, or around entire transactions, either from a transaction application client or application server perspective in different transaction models (flat or nested).

- *OperationAspect* extends the TransactionAspect with pointcuts for transaction operation, as shown in Figure 5-6. They provide a way for applications to capture arbitrarily complex operations; therefore, they define the sequence of operations that comprise the transaction body. This aspect defines pointcuts to demarcate the transaction operation scope, namely *BeforeTransactionOperation* and *AfterTransactionOperation.* The BeforeTransactionOperation creates an OperatoinJP and instantiates an OperationContext, as shown in Figure G-2 (appendix G). It exposes the before-operation event joinpoint to the OperationJP and then adds the OperationContext to the ContextJPRegistry. The AfterTransactionOperation retrieves the matching OperationJP from the OperationContext for the current transaction in the ContextJPRegistry, and exposes the after-operation event joinpoint to the OperationJP. Developers can use this aspect to weave advice before, after, or around a transaction operation.

```
public abstract aspect TransactionOperationAspect extends TransactionAspect{
    private Logger logger = Logger.getLogger(TransactionOperationAspect.class);

    declare parents: TransactionOperationAspect extends TransactionAspect;

    public pointcut BeforeTransactionOperation(OperationJP _operationjp) :
execution(* TransactionOperationAspect+.beforeOperation(OperationJP)) &&
args(_operationjp);
    public pointcut AfterTransactionOperation(OperationJP _operationjp):
execution(* TransactionOperationAspect+.afterOperation(OperationJP)) &&
args(_operationjp);
...
}
```

Figure 5-6. Extended Parameterized-Pointcuts in TransactionOperationAspect

```
public abstract aspect LockAspect extends TransactionAspect {
         private Logger logger = Logger.getLogger(LockAspect.class);

         public pointcut BeginRequestlock(LockingJP _lockingjp): execution(*
LockAspect+.beginSetLocking(LockingJP)) && args(_lockingjp);

         public pointcut EndRequestlock(LockingJP _lockingjp): execution(*
LockAspect+.endSetLocking(LockingJP)) && args(_lockingjp);

         public pointcut HoldingResource(ResourceLockedJP _resourceLockedjp):
execution(* LockAspect+.hold(ResourceLockedJP)) && args(_resourceLockedjp);

         public pointcut ReleasingResource(ResourceLockedJP _resourceLockedjp):
execution(* LockAspect+.release(ResourceLockedJP)) && args(_resourceLockedjp);
     ...
}
```

Figure 5-7. Extended Parameterized-Pointcuts in LockAspect

- *LockAspect* is derived from the TransactionAspect and thereby inherits the locking and resource-locked pointcuts, as shown in Figure 5-7. It involves setlock-event and release-event joinpoints to associate and disassociate the specified resource to/from the target transaction. It defines pointcuts BeginRequestlock, EndRequestlock, HoldingResource, and ReleasingResource that crosscut *ConcurrencyControlJoinpointTracker* to establish the lock context. BeginRequestlock creates an instance of LockingJP, exposes BeginlockEventJP to it, instantiates a lock context, and then adds the context to the ContextJPRegistry, as shown in Figure G-3 (Appendix G). The EndRequestlock retrieves the matching LockingJP from the target LockContext in the ContextJPRegistry, and exposes the EndlockEventJP to the LockingJP when the request lock is granted or refused. The HoldingResource creates a ResourceLockedJP and exposes the hold-event joinpoint to it. It also retrieves the matching lock context from ContextJPRegistry and then adds the ResourceLockedJP to the lock context in ContextJPRegistry. The ReleasingResource retrieves the matching ResourceLockedJP from the target LockContext in the ContextJPRegistry and then exposes the release-event joinpoint to the ResourcelockedJP and ends the locked resource. Developers can use this aspect to weave advice before, after, or around the entire locking perspective.

In DTPSs, the nested and concurrent transactions may occur with multiple other hosts, i.e., transaction in progress, which are also involved in a multi-threaded process. The aspects can apply for a transaction and keep track of the multiple concurrent transactions by maintaining a collection of contexts. A context for each transaction is maintained in terms of its own current context and association with the in-progress transaction.

5.7 Design Patterns Perspective on the *TransJ* Implementation

TransJ implementation offers a collection of functions for dealing with reusable transaction crosscutting concerns. To provide reusable code without influencing its architecture, it is built using these design patterns: Strategy, Singleton and Template method. They allow programmers to extend and customize the *TransJ* functionality.

The *TransJ* implementation of the strategy pattern contains the same concrete strategies (joinpoints) and contexts [104]. It declares two markers called TransJP and Context that are used in the concrete aspects. The context has a ConcurrentHashmap to store the relationship between each context and its current joinpoint.

ContextJPRegistry is a singleton class providing the list of the contexts, which sets and gets the appropriate context for the target joinpoint and execute the relevant advice. This pattern provides reusable pattern code in terms of setting and getting the strategies in the transaction-related context. The TransJP provides the joinpoints for each context in the list, and all the joinpoints are from inheritance relationships. It means that the joinpoints extends TransJP class.

Furthermore, *TransJ* implementation provides generic advices in the base aspects that follow the template method pattern [104]. Therefore, the base aspects are implemented as abstract aspects to contain the actual implementation of the template advices and pointcuts. This allows developers to quickly adapt them to the specific needs of their application by reusing and integrating them into existing or new applications.

CHAPTER 6

REUSABLE AND APPLICATION-LEVEL ASPECTS

This chapter discusses the reusable and application-level layers in *TransJ* architecture and provides examples of transaction-related crosscutting concerns implemented with *TransJ*.

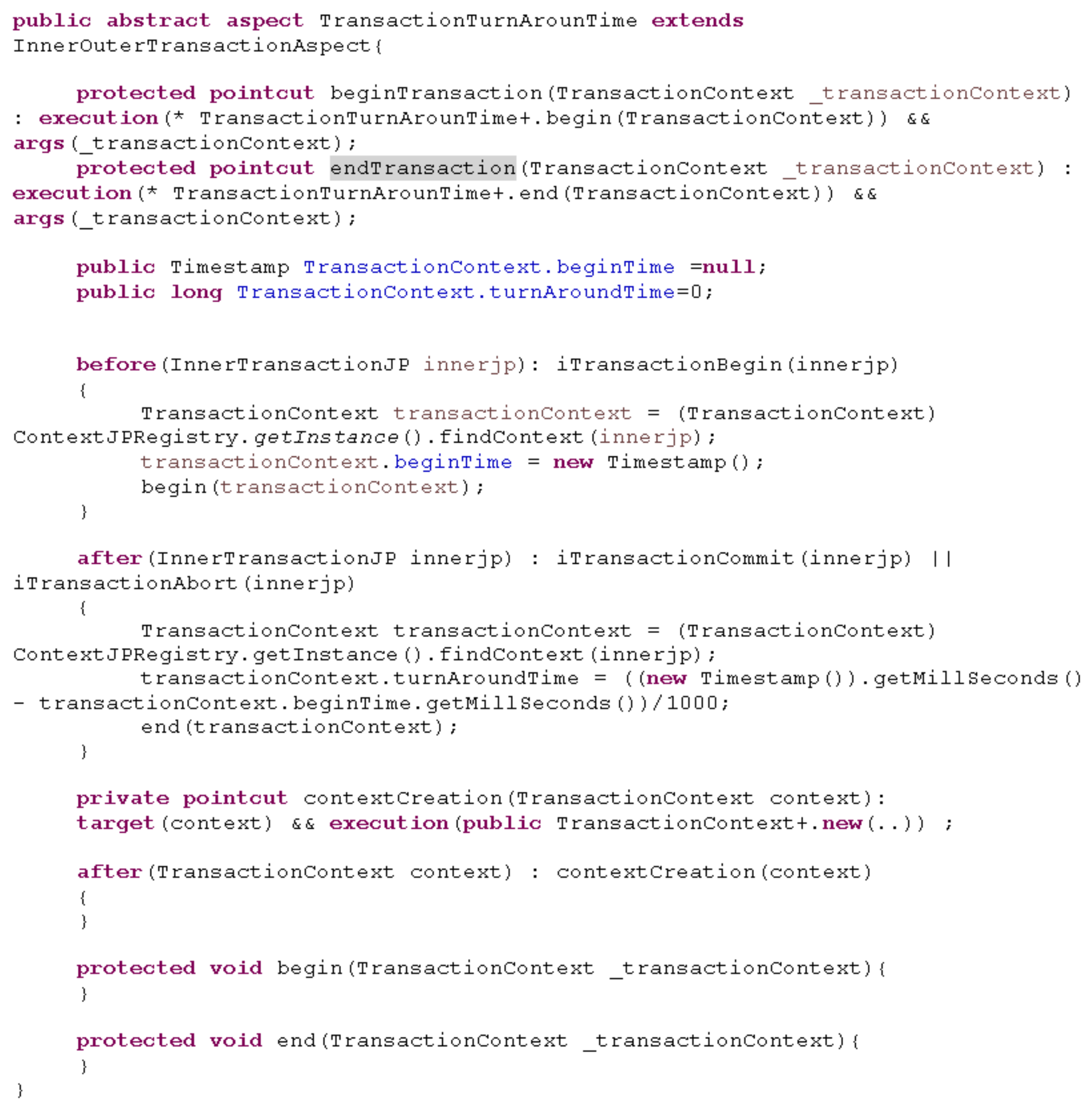
6.1 Reusable Aspects

Aspect developers implement reusable aspects by specializing the base aspects in *TransJ*. The reusable aspects represent general crosscutting concerns commonly found in distributed applications with significant transaction requirements. Table 1 lists the aspects currently in the reusable aspects library and Figure 6-1 shows part of the implementation of one of them. This is called the *TotalTurnAroundTimeMonitor*.

Table 1. Sample Reusable Crosscutting Concerns in Transactions

Aspect Name	Description
Optimizer	Tracks workload based on the most likely behavior of a transaction. It helps programmers to determine the most efficient way to execute a transaction by considering the possible behaviors on shared resources.
PerformanceAnalyzer	Helps the programmer to analyze the vast amount of transaction resource accesses for improving the application performance
Notification	Allows the developer to activate alarms for critical error, exceptions, time-based expiration, and invalid state.
Authenticator	Tracks consistent and secure transactions for handling authentication permissions in transaction domain.
AuditTrail	Records a history of actions executed by transactions and users. It includes a chronological list of steps that are required in order to begin a transaction, as well as bring it to completion. It records information such as who has accessed a transaction, what operation was performed on it, when it was performed, and how the state was changed.
LoggingByTransaction	Logs transaction context operations in a developer-defined format and domain.
JustInTime	Provides virtual helper methods for a transaction which help programmers share context information across hosts when necessary.
DeadlockAnalyzer	Detects transactions that are involved in the deadlock.
TotalTurnAroundTimeMonitor	Provides virtual helper methods for transactions which help programmers measure the responsiveness time by overriding their aspects in application level.

TransJ provides a library of reusable aspects for transaction-related crosscutting concerns, like *TransctionTurnAroundTime*, that helps programmers measure the responsiveness time. These aspects allow programmers to adapt the reusable aspects to new demands and to cope with the specific needs of their application by overriding these methods. Additionally, *TransJ* library provides other reusable aspects the make use of this and other reuse techniques to integrate them easily into existing or new applications. We



```
public abstract aspect TransactionTurnArounTime extends
InnerOuterTransactionAspect{

     protected pointcut beginTransaction(TransactionContext _transactionContext)
: execution(* TransactionTurnArounTime+.begin(TransactionContext)) &&
args(_transactionContext);
     protected pointcut endTransaction(TransactionContext _transactionContext) :
execution(* TransactionTurnArounTime+.end(TransactionContext)) &&
args(_transactionContext);

     public Timestamp TransactionContext.beginTime =null;
     public long TransactionContext.turnAroundTime=0;


     before(InnerTransactionJP innerjp): iTransactionBegin(innerjp)
     {
          TransactionContext transactionContext = (TransactionContext)
ContextJPRegistry.getInstance().findContext(innerjp);
          transactionContext.beginTime = new Timestamp();
          begin(transactionContext);
     }

     after(InnerTransactionJP innerjp) : iTransactionCommit(innerjp) ||
iTransactionAbort(innerjp)
     {
          TransactionContext transactionContext = (TransactionContext)
ContextJPRegistry.getInstance().findContext(innerjp);
          transactionContext.turnAroundTime = ((new Timestamp()).getMillSeconds()
- transactionContext.beginTime.getMillSeconds())/1000;
          end(transactionContext);
     }

     private pointcut contextCreation(TransactionContext context):
     target(context) && execution(public TransactionContext+.new(..)) ;

     after(TransactionContext context) : contextCreation(context)
     {
     }

     protected void begin(TransactionContext _transactionContext){
     }

     protected void end(TransactionContext _transactionContext){
     }
}
```

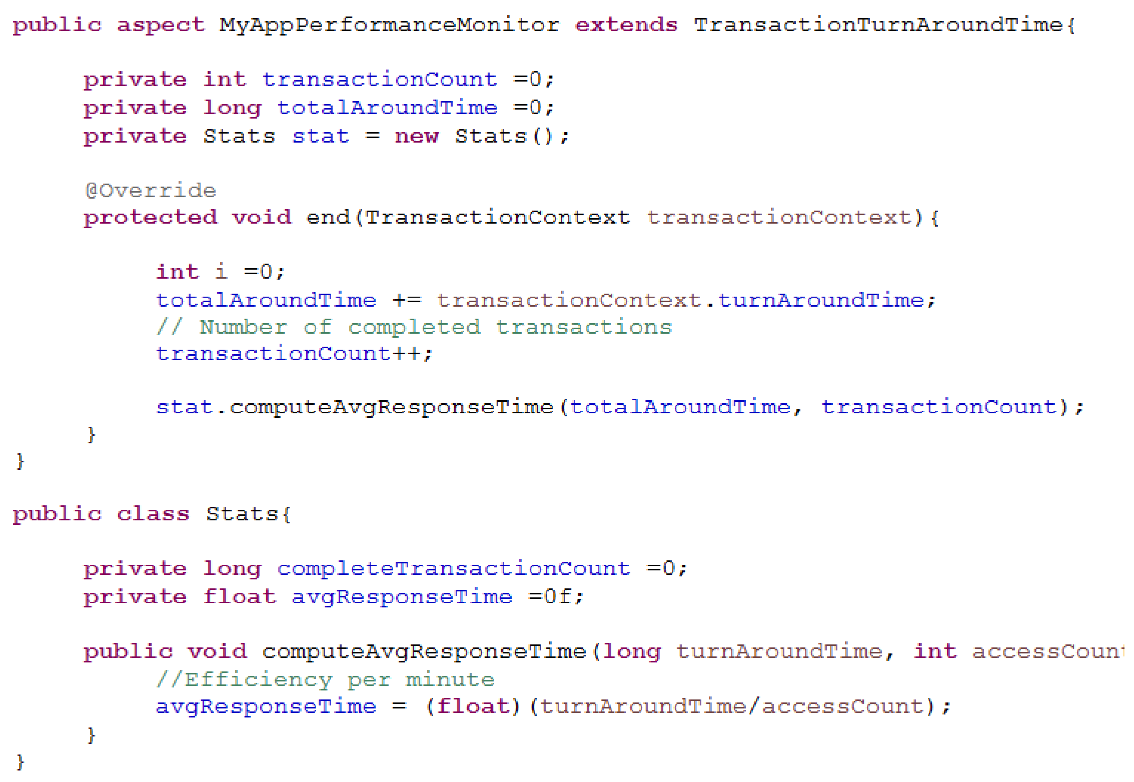
Figure 6-1. A Reusable-Aspect Code Snippet of *TransactionTurnAroundTime*

expect that reusable aspects will continue to grow as new generally-applicable transaction aspects are discovered, implemented, and documented.

6.2 Transaction Turnaround Time (TTT) Aspect in Reusable and Application Layers

TransJ allows reusable aspects to run a set of aspects to access context information dynamically. The reusable aspects use the base aspects to represent the abstract aspects that contains the template advice as discussed above.

As an example, this section describes the implementation of an application-level aspect that weaves performance measurements in the distributed transaction applications. For discussion purposes, assume that the performance measurements are a throughput and average-transaction response turnaround time statistics. In other words, it measures some performance-related statistics for transaction-based applications between a client and server, such as turnaround time (i.e., response time). Also, assume that the core application



```
public aspect MyAppPerformanceMonitor extends TransactionTurnAroundTime{

     private int transactionCount =0;
     private long totalAroundTime =0;
     private Stats stat = new Stats();

     @Override
     protected void end(TransactionContext transactionContext){

          int i =0;
          totalAroundTime += transactionContext.turnAroundTime;
          // Number of completed transactions
          transactionCount++;

          stat.computeAvgResponseTime(totalAroundTime, transactionCount);
     }
}

public class Stats{

     private long completeTransactionCount =0;
     private float avgResponseTime =0f;

     public void computeAvgResponseTime(long turnAroundTime, int accessCoun
          //Efficiency per minute
          avgResponseTime = (float)(turnAroundTime/accessCount);
     }
}
```

Figure 6-2. Performance Measure Crosscutting Concern

considers a transaction to be the completion a set of sub-transactions. Consider a transaction involving three sub-transactions. So, we can measure throughput for a unit of time, say 1 minute, by simply counting the number of these transactions completed in that minute. The average response turnaround time is the average of time spans from transaction begin times to transaction commit or abort times.

First, notice how this advice is derived from InnerOuterTransactionAspect and in doing so, it can reuse its implementation of the transaction turnaround time concept directly.

Figure 6-2 shows the key pieces of code for an aspect that implement this performance measure crosscutting concern. This snippet of code presents the implementation of measuring the total turnaround time and throughput for a nested transaction at the application level. As mentioned, the developers can implement and add application-level aspects into core application logic by reusing reusable aspects or extending base aspects in *TransJ*.

Second, notice how the aspect is derived from TransactionTurnAroundTime aspect and in doing so, it can reuse its implementation of the transaction turnaround time concept directly. Then, it adds some additional behavior at the end of a transaction to compute the average of the transaction responsiveness time per minute, i.e., efficiency.

6.3 Audit Trails Crosscutting Concerns

This example discusses the design and implementation of an aspect that can manage the audit trail for recording a history of actions executed by transactions and users. It will include a chronological list of steps that were required to begin a transaction as well as bring it to end. Imagine that the developer wants to make the *Conference Registration System* (CRS) as an auditable system that satisfies all the requirements to provide a comprehensive and thorough audit trail by implementing various levels of it, e.g., at the transaction level and operational level, as shown in Figures 6-3 and 6-4. This way allows developers to define audit trail, which records user registration and deregistration activities, as well as what names were issued by the attendees to the register for the conference. In other words, the audit trail concern keeps track of who did the transaction, to what (register or deregister), and when they did it, as well as who tried to do something but was unsuccessful. It’s useful for detecting conflicted registration, establishing a culture of responsibility and

```
public aspect TransactionAuditTrailAspect extends InnerOuterTransactionAspect{

     before(InnerTransactionJP innerjp): iTransactionBegin(innerjp){
          TransactionContext transactionContext = (TransactionContext)
ContextJPRegistry.getInstance().findContext(innerjp);
          // code to add a new record of transaction activities being begun
          ...
     }

     after(InnerTransactionJP innerjp): iTransactionBegin(innerjp){
          TransactionContext transactionContext = (TransactionContext)
ContextJPRegistry.getInstance().findContext(innerjp);
          // code to check and update the record of transaction activities and
status of the current transaction
          ...
     }
}
```

Figure 6-3. An Audit Trail Aspect at Transaction Level

accountability, reducing the risk associated with inappropriate registrations, detecting new threats and intrusion attempts, and identifying potential problems, etc. In transaction contexts, the audit trail aspect allows a transaction to create a monitor that encapsulates valuable information about the transaction and current transaction operations: in the Conference Registration System contains the information is usually the identity of a transaction, the name of the object, access type of the invoked method, operation arguments, etc.

The application-level audit trail aspects in Figures 6-3 and 6-4 extend aspects discussed in Section 5.7. On beginning the transactions, InnerOuterTransactionAspect ensures that it is beginning the transactions and starting to provide documentary evidence of the sequence of activities that have affected at any time a

```
public aspect  OperationAuditTrailAspect extends TransactionOperationAspect{

     Object around(OperationJP operationjp):
AfterTransactionOperation(operationjp){

          OperationContext operationContext =
(OperationContext)ContextJPRegistry.getInstance().findContext(operationjp);
          //code to check and update the record of transaction activities and
status of the operations
     }
}
```

Figure 6-4. An Audit Trail Aspect at Transaction Operation Level

registration or deregistration transaction. On transaction operations, TransactionOperationAspect gives a step-by-step documented history of the transaction and keep an eye on the transaction operations.

We conclude, the audit trail aspect can be used in any context: transaction context, operation context, or lock context, where there is a need to recall transactions, operations or any other meta-data about the target transaction that have been applied within a specific context. The information pulled by this aspect can, for instance, be beneficial for logging.

6.4 Optimizing Data Sharing

This section discusses the design and implementation of data-sharing optimizer aspect, i.e., Shared Aspect, in the context of a DTPS that collects data from different threads that run on distributed hosts, referred to here as a Gadget Manufacturing System (GMS) as described in Chapter 2 and Appendix A. Threads running synchronously within the same transaction may concurrently execute conflicting transaction operations that may cause failure or delay in the execution of transactions.

The shared aspect involves managing the shared resources by collection transactions in the GMS. The aspect manages the optimizing data sharing through monitoring and adjust the level of execution between distributed transaction operations. In other words, in order to optimize throughput and hence improve performance, Shared needs the context information of each operation of the transaction in many contexts. Therefore, a transaction that is accessed by a thread that is part of a context should notify the context of the upcoming access. That way, the context can take actions to implement the specified rules. Each accessed shared resource should be uniquely identifiable in a context of the target transaction. In *TransJ*, the ContextJPRegistry provides this functionality by keeping a list of all transaction and operation contexts that have interacted with the target transaction. Therefore, it is able to track all threads that have run with the transaction operations.

Figure 2-4 shows the multiple thread access to widget pile concurrently, and then turned them into new widget type by identifying the right transaction operations. Once the system notices the number of gadgets exists excess in the inventory, then goes to sleep to slow down the production speed, unless it again notices the amount of gadgets less than expected, then turn the system to increase the speed of production.

```
public abstract aspect SharedAspectMonitor extends LockAspect{

     String accessMode ="";

     void around(LockingJP lockingjp) : EndRequestlock(lockingjp){

          TransactionContext transactionContext =
ContextJPRegistry.getInstance().findTransactionContextbyJPId(lockingjp.getTid(
));
          OperationContext operationContext =
ContextJPRegistry.getInstance().findOperationContextbyJPId(lockingjp.getTid())
;

          Xid tid= transactionContext.getTid();
          Transaction transaction =
transactionContext.getInnerJP().getTransaction();
          Method method = operationContext.getOperationjp().getMethod();
          accessMode = lockingjp.getBeginlockEventjp().getLockMode();

          proceed(lockingjp);
          String lockResult = lockingjp.getEndlockEventjp().getLockResult();

          getsharedlock(tid, transaction, method, accessMode);
     }

     void around(ResourceLockedJP _resourceLockedjp):
ReleasingResource(_resourceLockedjp){
          TransactionContext transactionContext =
ContextJPRegistry.getInstance().findTransactionContextbyJPId(_resourceLockedjp
.getTid());
          OperationContext operationContext =
ContextJPRegistry.getInstance().findOperationContextbyJPId(_resourceLockedjp.g
etTid());

          Xid tid= transactionContext.getTid();
          Transaction transaction =
transactionContext.getInnerJP().getTransaction();
          Method method = operationContext.getOperationjp().getMethod();
          proceed(_resourceLockedjp);

          releaseSharedLock(tid, transaction, method, accessMode);

     }

     protected void getsharedlock(Xid tid, Transaction transaction, Method
method, String accessMode){
     }

     protected void releaseSharedLock(Xid tid, Transaction transaction, Method
method, String accessMode) {
     }
}
```

Figure 6-5. An Aspect Code Snippet of SharedAspectMonitor

In *TransJ*, Figure 6-5 shows implementation one of the reusable aspect, called SharedAspectMonitor. It creates a monitoring transaction operation, which manages the shared operations based on type of access. This aspect monitors the transaction operation context that contains information about the current operation thread and kind of access. In this example, developers can extend SharedAspectMonitor and override its methods to control the production process. In addition, we notice the developer can add some additional behavior to the SharedAspectMonitor by overriding the getsharedlock and releaseSharedlock to manage the shared resources over distributed transactions, as shown in Figure 6-6.

```
public aspect WidgetProductionSpeedControl extends SharedAspectMonitor{

     @Override
     protected void getsharedlock(Xid tid, Transaction transaction, Method 
method, String accessMode){
          // sleep the production process


          ...
     }

     @Override
     protected void releaseSharedLock(Xid tid, Transaction transaction, Method 
method, String accessMode) {
          // speed up the production process


          ...
     }

     public void speedControl(Method mehtod){
          // code to check the number of available
          //gadgets and add logic of production speed control
          // then make a decision to speed up or slow down
          ...
     }
}
```

Figure 6-6. A Code Snippet for WidgetProductionSpeedControl at Application Level

CHAPTER 7

MEASURING REUSABILITY, PERFORMANCE AND SOFTWARE DEVELOPMENT EFFICIENCY

This research aims to demonstrate the feasibility and utility of using *TransJ* and the reusable aspect library to implement transaction applications and transaction aspects of those applications. In Chapter 6, we discussed the applications that are implemented using the *TransJ* to provide evidence of improvement in reusability and performance. To investigate this contribution, we adapt a measuring method based on a metrics suite that extends the metrics traditionally used with the OOP, AOP, and later used in the *CommJ* research. Specifically, to measure the effectiveness of *TransJ* in comparison with AspectJ, we adapt and extend the *Extended-Quality Model* [30] and *Comparison Quality Metrics* (Sant'Anna quality model) [54][55] to include quality factors and internal attributes that have not been included in these models, forming the *Extended-Quality Model for Transactional Application* (EQMTA).

EQMTA consists of four elements: *Qualities*, *Factors*, *Quality Attributes*, and *Metrics*. See Figure 7-1. The qualities, i.e., reusability and performance, are the most abstract concepts in the model and represent the ultimate goals of "good" software. Each quality is affected by one or more factors, which are in turn determined by quality attributes (internal attributes). The quality attributes describe the internal view of the system attributes with a set of quality metrics that are defined and used to provide a scale and method for measurement.

Figure 7-2 show the specific qualities, quality factors, and quality attributes of the EQMTA's suite, and Figure 7-3 show the metrics. A single star (*) next to an element in either of these figures tags a concept that not exist in the original EQM [30] or Comparison Quality Model [55]. Double stars (**) mark elements that are in the previous models, but have been modified to be a measure quality in transaction systems.

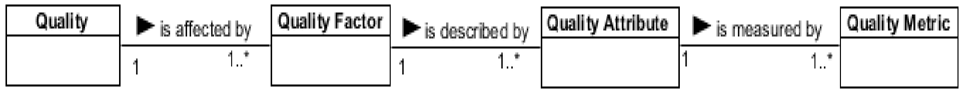


Figure 7-1. Elements of the Quality Model

7.1 Qualities

Qualities are the highest level abstractions that we want to primarily observe in our system. We picked reusability and performance as the important qualities to consider initially because of potential for cost savings that they both represent. The following qualities focus of our experimental research:

- *Reusability*. It is the ability of transaction elements to serve for structure of different elements in the same software system or across different ones. In other words, it is modifying existing

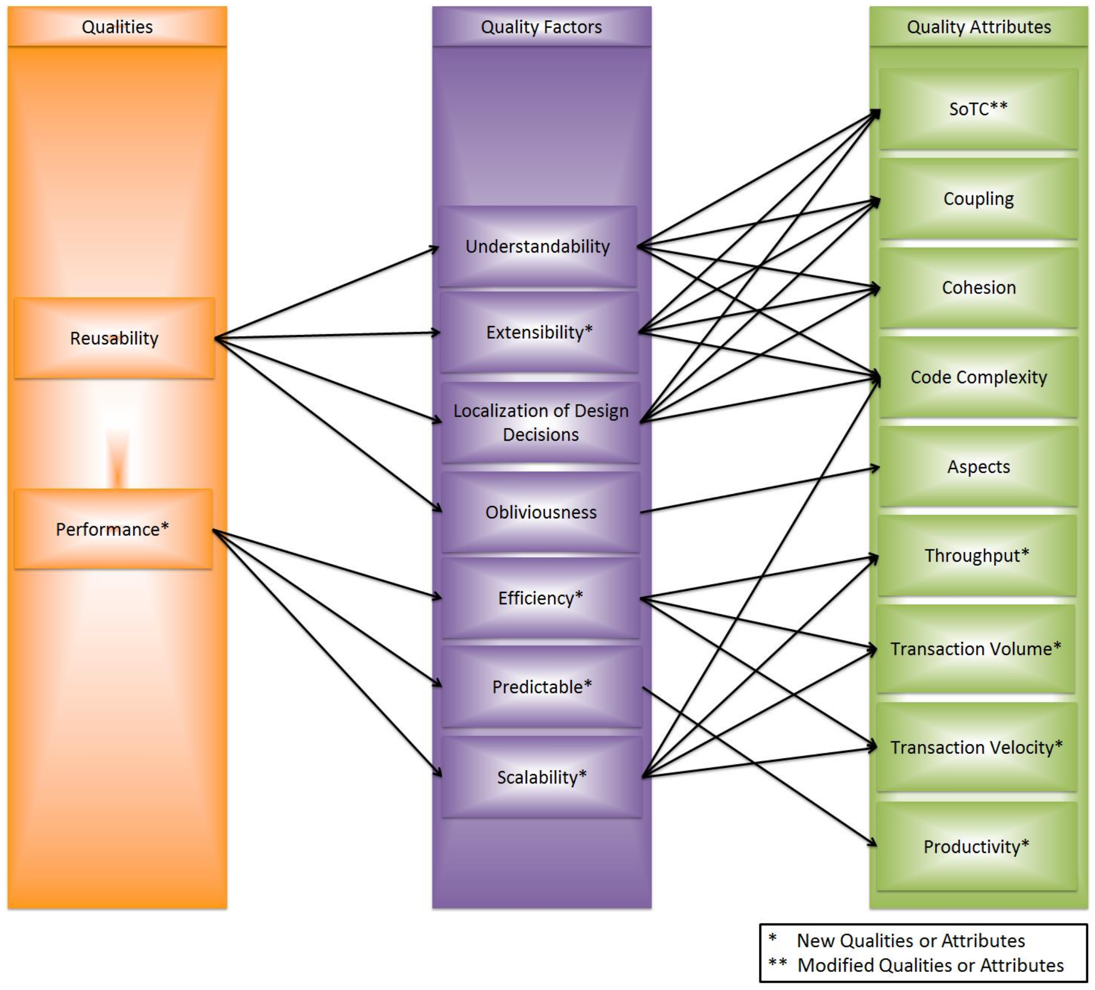


Figure 7-2. Extended-Quality Model for Transactional Applications (EQMTA)

components as needed to meet specific system requirements. The developers can use it to create separate components and organize the functionality of a system into independent modules.

   - *Performance*. It is characterized by the amount of useful work accomplished by a transaction system compared to the transaction element, time, and resources used.

7.2 Quality Factors

The quality factors are the secondary quality attributes that influence the defining primary qualities. These attributes are useful for the promotion of performance as well as reusability, because they are associated with well-established internal quality attributes of the software systems. Following are a list of the elements of our quality model:

- *Understandability:* It indicates the level of difficulty for studying and understanding a transaction system's design and code.
- *Extensibility:* It indicates the level of difficulty to expand the system's capabilities, and facilitate systematic reuse with a minimum amount of effort, in addition to make drastic changes to components in a transaction system without any need to change others.
- *Localization of Design Decisions:* It indicates the level of information hiding for a component's internal design decision. Hence, it is possible to make material changes to the implementation of a transaction component without violating the interface [7].
- *Obliviousness:* It is a special form of low coupling wherein base application functionality has no dependencies on crosscutting concerns [30].
- *Efficiency:* It indicates the level of performance of the software. In other words, it captures the ability of a transaction system to provide appropriate performance in relation to the amount of resources used, under stated conditions, such response time and throughput.
- *Predictability:* It indicates the ability to anticipate the amount of work the team can feasibly commit to providing exactly what the customer expects and delivering the value one time without error or delay.
- *Scalability:* It indicates to the capability of a system to increase its total throughput under an increased load when resources are added that refer to the increasing amount of transactions in a

capable manner to be enlarged to accommodate that growth. Therefore, it expresses aspects of the design that should be tuned for efficient transaction at any given scale.

Localization of design decisions, and code obliviousness were part of *Extended-Quality Model* [30]. We use them in our quality model due to the following reasons. Firstly, Raza [30] in his *CommJ* paper proposes three important characteristics of modular code, namely understandable, obliviousness and localization of design decisions. Hence, reasoning reusability in terms of understandability, localization of design decisions, and obliviousness are not complete. Introduction of efficiency, predictability, and scalability are also equally important. Secondly, by the time Parnas [62] and Coady [65] proposed that the definition of reusable modular code, obliviousness and extensibility 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, extensibility, efficiency, predictability, and scalability become very critical.

7.3 Quality Attributes

Quality attributes, i.e., internal attributes, do not rely on software execution and can therefore be measured statically. They are characteristics of software systems related to well-established software-engineering principles, which in turn are necessary to the achievement of the qualities and their respective internal factors. Following are the internal attributes in our EQMTA.

- *Separation of Transaction Concerns* (SoTC): It defines the ability to identify, encapsulate and manipulate unnecessary complexities of transaction system that are relevant to a particular concern [67].
- *Coupling (dependency)*: It is an indication of the strength of interconnections between the transaction components in a transaction system [68][100]. In other words, it measures the impact degree of AOP and *TransJ* mechanisms on reusability attributes, and collaborations between transaction components or between transactions and other system components.
- *Cohesion*: The cohesion of a component is a measure of the closeness of relationship between its internal components [54]. In other words, it measures the degree to which the pieces of a single module belong together. Low cohesion is associated with secondary concerns such as being difficult

to reuse and maintain. Cohesion is usually compared with coupling. High cohesion often correlates with loose coupling, and vice versa [68]. Low coupling is often an indicator of a well-structured transaction system and a good design, and when combined with high cohesion, supports the general goals of high reusability.

- *Code Complexity*: It measures how transaction components are structurally interrelated to one another. It indicates to the degree of difficulty in the transaction system's design and code [70].
- *Tangling*: It occurs when a single transaction component includes functionality for two or more concerns, and those concerns could be reasonably separated into their own components.
- *Scattering*: It occurs 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.
- *Aspects/Obliviousness*: It is an indication of the ability of aspects component to encapsulate and manipulate a crosscutting concern of the transaction system. It physically measures the number and length of a software system's aspects [30].
- *Throughput*: It is an indication of the capability of the system to complete a transaction within a specific interval. In other words, it is the rate at which transactions are processed by the system. It physically measures the amount of time is taken to respond to the request of a transaction.
- *Transaction Volume:* It is an indication of the efficiency of transaction system to handle huge data volume, which determine the amount of transactions processed by the system over the defined period of time, such as the committed transactions, aborted (uncommitted), and timed-out transactions during the transaction system execution.
- *Transaction Velocity:* It gives an indication of the performance of the transaction system, it refers to how fast a transaction is processed within the context of accessing resources in transaction systems.
- *Productivity:* It is an indication of the amount of effort needed for understanding, programming and debugging the transaction system components. It physically considers the amount of bugs, and total development time into active and passive times.

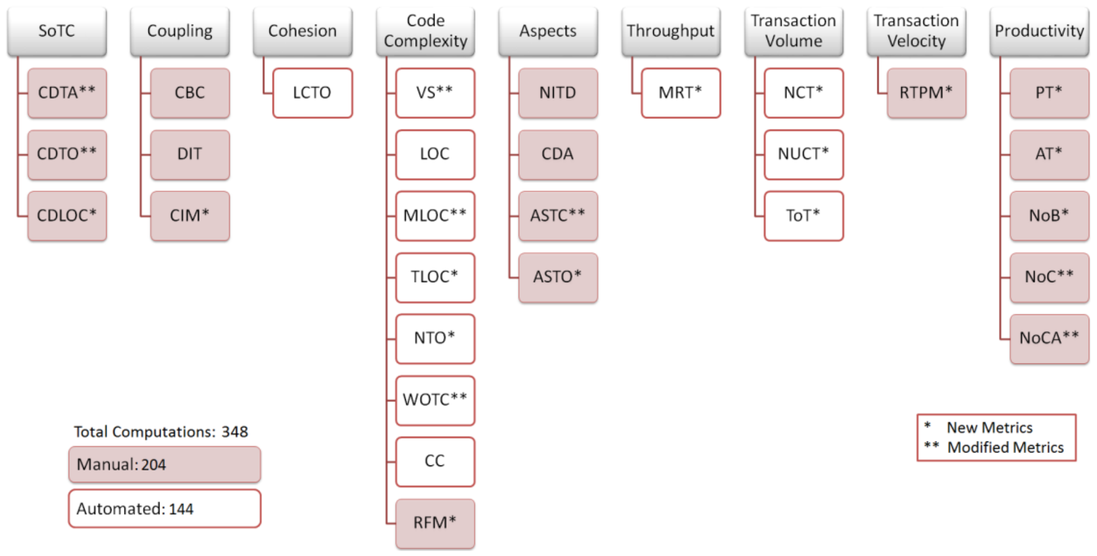


Figure 7-3. Measurement Metrics in EQMTA

7.4 Quality Metrics

Each internal attribute is related to a set of the proposed metrics. The proposed suite of metrics captures information about the design and code in terms of fundamental software internal attributes as discussed above.

Figure 7-3 presents the metrics that EQMTA uses to measure each of the internal attributes. We have tailored the definition of these metrics to reflect the new abstractions introduced by aspects in terms of quality transaction system attributes. The EQMTA is composed of 29 design and code metrics. In the following subsections, these metrics are congregated according to internal attributes that are measured. The description of each metric emphasizes how it satisfies our measurement requirements. The relevance of these metrics for reuse and performance is discussed in our experimental procedural (see Chapter 9). Twelve of the metrics can be computed automatically from the code written by the subjects. The others have to be computed by hand.

7.4.1 *Separation of Transaction Concern (SoTC) and Scattering Metrics*

The EQMTA defines the *Concern Diffusion in Transaction Application* (CDTA), *Concern Diffusion over Transaction Operations* (CDTO) and *Concern Diffusion over Line of Code* (CDLOC) to measure the degree to which a single concern in transaction system maps to transaction components in the system design and code, where

- CDTA counts the number of primary transaction components (classes or aspects) whose main purpose is to contribute to the implementation of a single transaction-related concern. In other words, it counts number of advices and operations that access the primary transaction components to gather relevant context information by using them in attributes declaration, formal parameters, return types, introductions (inner-type declarations), method call, and throw declarations.
- CDTO counts the total number of primary transaction operations and advices whose main purpose is to contribute to the implementation of a single transaction-related concern. In other words, it counts the number of methods and advices that access any primary transaction component to pull all relevant operation context information by calling their methods or using them in formal parameters, local variables, return types, and throws declarations. Constructors also are counted as operations.
- CDLOC counts the total lines of primary transaction components (a class or aspect) whose main purpose is to contribute to the implementation of a single transaction-related concern. It counts the total number of occurrences that access the primary transaction-related concern information, i.e., transaction contexts, by using them in attribute declarations, formal parameters, return types, throw declaration, local variables, inter-type declarations, or method calls. This considers the total number of transition points for each concern through the line of code over transaction component. Transition points are points in the code where there is a concern switch. The use of this metric requires a shadowing process that partitions the code into shadowed areas and non-shadowed areas. Shadowed areas are lines of code that implement a given concern. Figure 4-1 presents transition points (events) in our implementation. Transition points are the points in the transaction code where there is a transition from a non-shadowed area to a shadowed area and vice-versa [55].

The higher the CDTA, CDTO, and CDLOC, the more intermingled is the concern code within the implementation of the transaction-related components. Otherwise, the lower the CDTA, CDTO, and CDLOC, the more localized is the concern code.

7.4.2 *Coupling Metrics*

The EQMTA measures coupling from different viewpoints. It defines the *Coupling between Components* (CBC), *Depth Inheritance Tree* (DIT), and *Coupling on Intercepted Modules* (CIM), where

- CBC is defined for counting the number of other transaction components to which a class or an aspect is coupled. The CBC for a transaction component, it counts the total number of classes or aspects declared in its attribute declarations. For each transaction operation, it counts the number of classes referenced by formal parameters, return types, throws declarations, introduction, and local variables from which shared context information are made. CBC for aspects also includes the number of components referenced by aspect introductions, component referenced by pointcuts, and components referenced by each advice. CBC counts each component only once. The low value of the CBC is desired [55].
- DIT: It counts how far down in the inheritance hierarchy a class or aspect is declared. Increase in the growth of the DIT, the lower-level components inherit or override many methods. This leads to hitches in understanding the code and design complexity when endeavoring to predict the behavior of a component.
- CIM: It counts the number of classes, aspects or interfaces explicitly named in pointcuts of a given aspect. It indicates the direct knowledge an aspect has of the rest of the system. High values indicate tight coupling, due to high crosscutting.

The larger the numbers for CBC, DIT or CIM become, the more difficult it is to understand the system. Extreme coupling amongst components is harmful to modular design and prevents reuse, because the higher the sensitivity to changes in other parts of the design. On the other hand, the lower the coupling for more independent components becomes, it easier it is to reuse it in another application.

7.4.3 *Cohesion/Tangling Metrics*

The EQMTA defines the following metrics for measuring cohesion and tangling among components: *Lack of Cohesion in Transaction Operations* (LCTO).

LCTO measures the lack of cohesion of a class or aspect in terms of the amount of method and advice pairs that do not access the same instance variable and hence should be separated [69]. If the related transaction methods do not access the same instance variable, they logically represent unrelated components. In other words, the lower value of LCTO for a more independent transaction component becomes, it implements a single logical function.

7.4.4 *Code Size and Complexity Metrics*

The EQMTA measures code complexity from different viewpoints. It defines metrics that are concerned with the different aspects of the system complexity. Size metrics measure the length of a software system's design and code. The following a list of size and complexity metrics: *Vocabulary Size* (VS), *Line of Code* (LOC), *Method Lines of Code* (MLOC), *Transaction Lines of Code* (TLOC), *Number of Transaction Operations* (NTO), and *Weighted Operations per Transaction Component* (WOTC), *McCabe's Cyclomatic Complexity* (CC), and *Response for Module* (RFM).

- VS: It counts the number of classes and aspects into the system. Sant' Anna mentioned that if the number of components increases, it is a clue of more cohesive and less tangled set of aspects [55].
- LOC: It counts the number of physical lines of active code (executable lines) that are in the software. Size can be measured in a variety of ways. These include counting all physical lines of code, or the number of statements. The greater the LOC, the more difficult it is to understand the system and harder find the lines that must be changed during evolution activities or understand the implementation of the required functionalities during reuse activities [69].
- MLOC: It counts the method lines of code, often omitting comments and/or omitting blank lines. In [69], Kremer claimed that the greater the average of MLOC for a component, the more complex the component would be.
- TLOC: It counts the transaction lines of code. This is the measure of the size of a transaction. Documentation and implementation comments as well as blank lines are not interpreted as code.

- NTO: It counts the number of operations in a transaction component. A transaction contains a large number of operations are less likely to be reused. Sometimes LOC is less, but NTO is more, which indicates that the transaction component is more complex.
- WOTC: It measures the complexity of a transaction component in terms of its operations. WOTC does not specify the advices and methods complexity measure, which should be tailored to the specific contexts. On the other words, it is a sum up the complexity of each advices and methods of aspects and class (WOTC = complexity of operation1 + complexity of advice1 + ... + complexity of operationN). The transaction operation complexity degree is obtained by counting the number of parameters of the transaction operation, assuming that a transaction operation with more parameters than another is likely to be more complex and less understandable [30][55]. The number of advices and methods and complexity is an indication of how much time and effort is required to develop and maintain the transaction-related components. The larger the value of weighted operations, the more complex the program would be [69].
- CC is a quantitative measure of the complexity of programming instructions. It is intended to measure system complexity by examining the software program's flow graph [70] [71]. In practice, CC amounts to a count of the decision points present in the software system. It can be calculated as:

$$CC = E - N + 2P \qquad \text{Equation (1)}$$

where,

  - E is the number of edges of the graph.
  - N is the number of nodes, and
  - P is the number of discrete connected components (nodes).

CC measures the logical complexity of the program. CC is originally intended as a measure of the number of the test case space. In other words, it defines the number of independent branches 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 [70]. The high value of CC affects program reuse.

- RFM: It counts the number of methods and advices that are executed by a given transaction in response to the request received by another transaction or system. Transactions with a higher RFM value are more complex and complicated. A lower value of RFM is more desired.

7.4.5 *Aspect/Obliviousness Metrics*

The EQMTA involves metrics on concerns that evolve into concrete pieces of code, i.e., Aspects, and contribute directly to the core functionality of the transaction software system. This model defines the following aspect metrics: *Number of Inter-type Declarations* (NITD), *Crosscutting Degree of an Aspect* (CDA), *Aspect Scattering over Transaction Components* (ASTC), and *Aspect Scattering over Transaction Operations* (ASTO).

- NITD: It counts the number of inter-type declarations in the aspects and the number of times they are used, which also includes their reference in the aspects and application classes. A higher value of NITD indicates a tighter coupling between the aspect and application components.
- CDA: It counts the number of modules affected by the pointcuts and by the introduction in a given aspect. CDA implies tight coupling between the modules which indicates complexity [68].
- ASTC: It counts the number of aspect components scattered over transaction application components. It measures the tangling of aspects in the application components. The more tangling of aspects in the program makes the original transaction application less reusable.
- ASTO: It counts the number of aspect components, i.e., advices and methods, scattered over transaction application operations. ASTC gives a high-level overview of the application tangling in the aspect components, but ASTO provides more insight on operations-level tangling of transaction applications inside aspect components.

7.4.6 *Transaction Throughput Metrics*

The EQMTA defines the rate of the *Mean Response Time* (MRT) to measure the performance of an individual transaction, in milliseconds.

- MRT counts the total time taken between the submission of a transaction request for execution and the return of the complete output for the front-end user. On the other words, it is the amount of time required for transaction completion, i.e., commit or abort. The response time for a transaction tends to decrease as you increase overall throughput.

7.4.7 *Transaction Volume Metrics*

The EQMTA defines the following transaction volume metrics: *Number of the Committed Transactions* (NCT), *Number of the uncommitted (aborted) Transactions* (NUCT), and *Timed-out Transaction* (ToT).

- NCT counts the total number of committed transactions within an execution interval, i.e., a second.
- NUCT counts the total number of the aborted transactions within an execution interval, i.e., a second.
- ToT counts the total number of the timed-out transactions within an execution interval, i.e., a second.

7.4.8 *Transaction Velocity Metrics*

The EQMTA defines the *Rate of the Transaction Per Minute* (RTPM) metrics to measure the velocity of the transaction.

- RTPM refers to the average speed value of transaction processing. In other words, the average number of transactions that are begin completed, either committed, aborted, or timed-out, per minute on the transaction system.

The transaction rate can be expressed as transactions per minute, therefore, we can then calculate the transaction rate for system as follows:

$$Transaction\ Rate = \frac{Transactions}{60\ s} \qquad \text{Equation (2)}$$

7.4.9 *Maintenance History Metrics*

The EQMTA defines productivity metrics to measure development efficiency in terms of the time needed to maintain the transaction applications. We divide total productivity (maintenance history) into *active time* (AT), *passive times* (PT), *a number of bugs* (NoB), *a number of changes in concern at the application level* (NoC), *a number of changes in concern and its application* (NoCA). The *active time* will be spent on typing and producing actual code, whilst the *passive time* is spent on reading and understanding the source code, looking for bugs, etc. These metrics are used to predict the amount of effort that will be required to develop a transaction program. A lower value of PT, AT, NoB, NoC and NoCA is more desired and will drastically increase the efficiency of the development component of transaction software systems. Below, we list the maintenance history metrics: *Active Time* (AT), *Passive Time* (PT), *Number of Bugs* (NoBs), *Number of changes in the application and its concerns* (NoCA), and *number of changes in concerns* (NoC).

- AT calculates the time that is needed to develop different transaction software components, i.e., an average time to write code, e.g., aspects.
- PT calculates the time that is spent on the other activities which are concerned with the development of the transaction system, such as the time to read the code, understand code, detect errors and trace bugs.
- NoB counts the total number of bugs during the development time.
- NoC and NoCA count the number of changes required to reuse the concern for another application, and to maintain the concern, respectively. The difference among them is that the NoC only considers changes in the concern; however, the NoCA considers changes both in the concern and application.

CHAPTER 8

EXPERIMENTAL HYPOTHESES

The theoretical ideas that underpin *TransJ* lead to the eight concrete hypotheses [112]. All of these hypotheses have the same premise and refer to the metrics defined in the EQMTA described in Chapter 7.

- *Better Encapsulation and Separation of Concern (SoC)*.

  If transaction-related crosscutting concerns are effectively modularized and encapsulated in *TransJ* aspects, then the software has better SoCs and less scattering than equivalent systems developed with AOP design techniques, especially AspectJ.

- *Supporting a Loose Coupling*

  If transaction-related crosscutting concerns are effectively encapsulated in *TransJ* aspects, then the software has a lower coupling than equivalent systems developed with AOP design techniques, especially AspectJ.

- *Higher-Cohesion and Less Tangling*

  If transaction-related crosscutting concerns are effectively encapsulated in *TransJ* aspects, then the software has higher cohesion and less tangling than equivalent systems developed with AOP design techniques, especially AspectJ.

- *Reducing the software code size and complexity structure*

  If transaction-related crosscutting concerns are effectively encapsulated in *TransJ* aspects, then the software is not significantly larger or complex than equivalent systems developed with AOP design techniques, especially AspectJ.

- *Increasing the capacity of the software obliviousness*

  If transaction-related crosscutting concerns are effectively encapsulated in *TransJ* aspects, then the software is significantly more oblivious than equivalent systems developed with AOP design techniques, especially AspectJ.

- *Preserving the software efficiency*

  If transaction-related crosscutting concerns are effectively encapsulated in TransJ aspects, then the software would keep or improve runtime performance under stated conditions for a stated period of time, compared with equivalent systems developed with AOP design techniques, especially AspectJ (measure by Jboss framework functions).

- *Improving the capacity of software extensibility*

  If transaction-related crosscutting concerns are effectively encapsulated in *TransJ* aspects, then the extension part, i.e., crosscutting concern, of the software would require a smaller number of changes to reuse (measure by Eclipse IDE diff function) than equivalent systems developed with AOP design techniques, especially AspectJ.

- *Improving the productivity of the development process*

  If *TransJ* provides a better modularization of transaction-related crosscutting concerns, then the development transaction system would be less complicated and more readable. Thus software development efficiency would be increased, so that the system would be created faster than equivalent systems developed with AOP design techniques. In other words, the total programmer's working time should be shorter than the development time of analogous systems developed with AOP techniques, especially AspectJ.

To determine whether *TransJ* improves reusability without sacrificing performance, I conducted an experiment that tests these hypotheses. The next chapter describes this experiment.

CHAPTER 9

EXPERIMENTAL PROCEDURE

The experiment to test the hypotheses about *TransJ*'s benefits consists of the sixteen steps distributed over two phases as listed below [113]. Sections 9.1 through 9.7 provide additional details about the steps. Section 9.8 discusses the experiment's independent and dependent variables, Section 9.9 describes how I minimized threats to validity caused by extraneous variables, Section 9.10 discusses some restrictions and Section 9.11 describes how I computed the above hypotheses using the EQMTA metrics.

- The first phase of designing the experimental method involves:
    1. All the researchers passed the online Human Research Training course offered through the *Collaborative Institutional Training Initiative* (CITI). See APPENDIX F for additional details.
    2. All the researchers submitted an application for a Human Research Experiment to the *Institutional Review Board* (IRB) and got its approval. See APPENDIX F for more details.
    3. Developed three non-trivial software applications and documented their requirements, design, and implementation. See section 9.2 for more details.
    4. Selected three common transaction-related crosscutting concerns for the above sample applications and developed an initial requirements specification document. See Section 9.3 for more details.
    5. Sent invitation letters (See Appendix F.2) and recruited four developers who were experienced in object-oriented software development (Section 9.4), and randomly organized them into two study groups: 1 and 2.
        a. Group 1 programmed using an AOP technology.
        b. Group 2 programmed using a *TransJ* fashion.
    6. Had the volunteers complete a survey that assessed their background and skill levels.
       See APPENDIX C.
    7. Provided JTA, Arjuna, and JBoss Application Server training to developers in Group 1, and 2, and had them worked through some practice applications. See Section 9.5.

8. Provided AOP training to developers in Group 1, and had them worked through some practice applications. See Section 9.5.
9. Provided *TransJ* training to developers in Group 2, and had them worked through some practice applications. See Section 9.5.

- The second phase of designing the experimental method, which involves:

10. Gave three sample applications mentioned above, associated documentation, and initial requirements specifications of all three concerns to the volunteers (APPENDIX A and APPENDIX B).
11. Asked the volunteers to complete a pre-implementation questionnaire (Appendix D.1), once they understood the code and documentation provided them in Steps 7, 8, 9, and 10.
12. Asked the volunteers to develop the three crosscutting concerns, and then collected their implementations using Bitbucket repository, and Box-file sharing.
13. Asked the volunteers to complete a post-questionnaire that gathered some additional information to measure quality metrics (See Appendix D.2.)
14. Measured the quality metrics using EQMTA, collected findings from the logs and pre/post-questionnaires of all activities.
15. Evaluated the reusability and performance of the various software artifacts using EQMTA. See Section 9.7 for details on the metrics and experiment.
16. Interpreted the results of the experiment.

Section 9.8 summarizes the control, independent, dependent, and extraneous variables for this experiment. Section 9.9 describes potential threats to validity of the research experiment.

9.1 Experimental Approval

In the first step, we got a prior approval by the *Institutional Review Board* (IRB) [25]. To do so, we submitted an application for conducting this Human Research Experiment to the USU IRB and got its approval. Before submitting this application, all the researchers passed the online human research experiment-training course offered through the *Collaborative Institutional Training Initiative* (CITI) [66]. See APPENDIX F.

9.2 Selection of Sample Applications

To improve the veracity of the experiment, it is important that the selected sample applications are non-trivial transaction systems and that their transactions represent a wide variety of concerns. To this end, we selected three applications based-on various criteria as: they involved different type of transaction models and concepts; they were implemented on the JTA API, X/Open standards and run on the Jboss Application Server; they included shared resources; two of them were distributed over many application hosts; they were multithreaded; and they were the ones who belong to the common fields are based on the transactions. In addition, the applications were diverse in the way they implemented transaction, i.e., Flat transaction vs. nested transaction, and therefore provided good coverage of different types of transactions and transaction–related concepts. Table 2 shows that these applications are categorized according to some of characteristics such as:

Table 2. Categories of Selected Applications

Categories	**Selected Applications**	**Gadget Manufacturing System**	**Conference Registration System**	**Local Bank System**
1	***Distributed***	✓	✓	
	Local			✓
2	***Flat***		✓	✓
	Nested	✓		
3	***Many Resources***	✓		
	Few Resources		✓	✓
4	***High Concurrency***	✓	✓	
	Low Concurrency			✓
5	***High potential for Conflict***	✓		
	Low potential for Conflict		✓	✓

Table 3. Sample Selected Applications

Application Name	Description
Local Bank System	The programmer implemented a transaction application where a server would calculate the response time between local transactions provided by the client, over a JDBC connection.
Conference Registration System	Flat transactions access a single shared resource. The programmer implemented a simple audit trail tracker that maintains a record of system activity both by processes and by user actions of the system. This tracking can help to detect registration and other suspicious behavior. In conjunction with detecting registration violations, recording the date and time of each entry of information for each attendee, and preserving the original content of the recorded information when changed, updated or corrected.
Gadget Manufacturing System	Distributed concurrent transaction accesses distributed data shared amongst multiple transaction components in multithreaded environment, so if all the threads succeed it will commit else abort. The programmer implemented an optimization mechanism in order to optimize sharing of the distributed shared-resources across hosts.

- Distributed vs. Local
- Flat vs. Nested transaction
- Many resources vs. Few resources
- High Concurrency vs. Low Concurrency
- High Potential for Conflict vs. Low Potential for Conflict

Table 3 describes the selected applications. Appendix A provides additional details about them. Volunteers were provided with the application code along with the functional requirement documents and UML diagrams.

9.3 Identifying Crosscutting Concerns from Sample Selected Applications

Since the experiment would eventually require programmers to adapt or extend applications for requirements that represented transaction-related crosscutting concerns, our methodology included a step, which systematically selected our representative crosscutting concerns. Volunteers would have to apply each of these to the applications, individually. We picked three common transaction-related crosscutting concerns for the experience such that they were applied to all the sample applications and the various concepts of

Table 4. Selected Sample Crosscutting Concerns

Aspect Name	Description
Measuring Performance	It measures some performance-related statistics for transaction-based applications between a client and server, such as turn-around time (i.e., response time).
Data-Sharing Optimization	It shares context information across hosts only when necessary. It includes knowledge of contexts related to transactions to allow different transaction operations to access shared-resources across hosts.
Audit Trail	It records a history of actions executed by transactions and users in order to monitor transaction activities and provide assurance that meet the predefined minimum requirements.

transactions, as shown in Table 4. To reduce chaos in our data, we wanted to make sure that these crosscutting concerns were adequately simple to a novice developer, who meets the specific criteria that are listed in Section 9.4, could understand and integrate them into the selected sample applications in less than 15 hours, regardless of whether *TransJ* is used. Table 4 and APPENDIX B present the set of selected crosscutting concerns for the experiment.

9.4 Recruitment of Developers

To transparently recruit the developers, we sent invitation letters and recruited four volunteer developers who were experienced in object-oriented and aspect-oriented software development, Java, transaction, and software-engineering design principles such as reusability.

The selection criteria for hiring developers to experimental research was based upon criteria according to developer capabilities, and relevant to my research field. Ideally, all participants were under-graduate and graduate students in Computer Science major; they had taken courses in algorithms, programming language (i.e., Java), data-structure, database system, object-oriented software development, and software engineering. Additionally, we had who had studied distributed systems, AOP and the concepts of transaction management systems; and they had a good exposure of:

- *Unified Modeling Language* (UML),
- implementing at-least one project in Java, the size of the project is comparable to the scope of our implementations.

- implementing at-least one database-based application using any type of database management systems (e.g., SqlServer, MySql, Postgress, DB2, H2 or Access) using Java,
- implementing at-least one distributed application project, and
- implementing at-least multithreaded programming project.

We randomly organized them into two study groups: 1 and 2. Group 1 programmed using an AOP approach and Group 2 used *TransJ*. Next, the participants completed a survey that assessed their background and skill levels. We also provided JTA, Arjuna, Jboss, AOP training to developers in Group 1, and had them worked through some practice applications. Similarly, we trained Group 2 developers with *TransJ*, and had them worked through some practice applications.

We hired four developers only, each group comprised two developers because there is limited availability for developers that met the selection criteria.

9.4.1 *Invitation Letter*

Initially, we sent an invitation letter by the Computer Science Department to all under-graduate and graduate students. See Appendix F.2.

9.4.2 *Anonymous Identity*

To protect the privacy of participants, we used aliases. Once selected the required volunteers, all volunteers tagged with a specific unique identifier. Therefore, data and code gathered from the volunteers were tagged with this identity to hide the personal identifiable information. An additional safeguard is included in this experiment to maintain data and codes in a separate secure location using a shared file, like *Box* and *Bitbucket*. The *Box* is operated by *Utah State University* (USU). The *Bitbucket* is used to monitor, collect and manage changes to the source code. After experimenting, the identity of the volunteer is still hidden by deleting the records that may indicate the identity of individual volunteers.

9.4.3 *A Survey to Assess the Level of Volunteer Skills*

To ensure a high quality experiment, we identified the effects of extraneous variables (Section 9.9), volunteers were asked to fill a pre-placement survey after hiring and before starting the experiment to obtain information on their effectiveness in meeting the needs of the experiment and expectations. See APPENDIX C for more details. The results of this survey, provided in full in APPENDIX E, clearly indicated that our selection of candidates satisfied all the criteria that are mentioned above.

9.5 Training of Developers

After selecting the developers, we organized them into two groups (Group 1 and Group 2) and place them randomly in two categories (i.e., *AOP*'s category, and *TransJ*'s category). Group 1's developers were trained on how to write aspects using AspectJ, and the Group 2's developers were trained on how to write aspects using both AspectJ and *TransJ*. During training, each developer implemented three sets of examples, similar to those that would be a part of our experiment. Each group undergone a training program for the purpose of achieving the experiment as follows:

- JTA, Arjuna and Jboss Application Server Training: All developers were trained on how to configure the JTA, Arjuna and run the Jboss Application Server.
- AOP Training: Group 1's developers were trained on how to write aspects in AOP using AspectJ.
- *TransJ* Training: Group 2's developers were started with the tutorial so they could understand the basic principles and the architecture of the *TransJ*, and had them worked through some practice applications using its library. In addition, they were trained on how to plug-in the *TransJ* with AspectJ.

By the end of training, each developer filled the pre-implementation questionnaire (APPENDIX D). This kind of questionnaire revealed that all volunteers were in good understanding, coding and debugging the language related complications, such as EJP, JPA, Jboss Server, etc.

9.6 Implementing the Selected Crosscutting Concerns Using the Set of Requirements and Collected Artifacts

All volunteers were given an initial set of requirements in which they were asked to implement three transaction-related crosscutting concerns (Section 9.3) using the sample applications (Section 9.2). More details of the applications and their associated concerns that are discussed in Chapter 6, APPENDIX A and APPENDIX B, respectively.

During this phase, we analyzed the understanding of the requirements, familiarity with the language and tools, and debugging the most prominent challenges. They also recorded hourly journals of maintenance history.

By the end of implementation, each developer filled the post-implementation questionnaire (APPENDIX D). Observation of this questionnaire indicated that all developers correctly understood the requirements, familiarized with the language, tools, and debugged the challenges. On requirements understanding, 50% of the total developers agreed that understanding and analyzing the requirements properly was the most time consuming in all activities. 75% of the total participants agreed that familiarity with the language/tool, e.g., JTA, JBoss, Arjuna, was the hardest thing during the initial activity of implementation, whereas no participant raised this issue again in the second and third activities. On debugging for AspectJ took more time than *TransJ* development. Explicitly, 25% of the participants supported this observation in activity 1, and 75% supported it again in activity 2 and 3. This observation shows that debugging time may be more associated with the complexity of the requirements than to experience with the implementation environment. See Section 10.9 and Appendix E for more information and additional observations.

9.7 Measuring Dependent Variables using Reuse and Performance Metrics

We measured EQMTA code metrics using both manual-based and automated tool-based methods [22][72]. Total measurements include following: experiment input variables included a total of four developers and three applications with each; experiment generated a total of 12 software systems against which the metrics need to be applied; the 29 code metrics of EQMTA, which will have a total of 348 measurements. Of these, 144 measurements from 12 metrics will be generated using tools, and 204 measurements from 17 metrics will be calculated manually.

Once data collection using these code metrics measurement procedure, then we interpreted the hypothesis in Chapter 8 using different experiment variables (see Section 9.8).

9.8 Experiment Variables: Independent, and Dependent Variables

I had tight enough control in this experiment to ensure a good experience. We observed what happened in changing the implementation approach as an independent variable with a focus on the dependent variable to see how it responded to the changes in the independent variables. For this experiment, the independent variables corresponded to factors that represented the implementation method. In our research, we had two possible values: AOP, and *TransJ*.

The dependent variables were those that we want to observe possible difference changes occurred in an AOP's group and *TransJ*'s group implementation. All instruments in our EQMTA (see Chapter 7) represent our dependent variables. Quality measurement metrics (Section 7.4) were our direct independent variables. Quality attributes (Section 7.3) were indirect dependent variables, which were interpreted from measurement metrics. Factors (Section 7.2) were indirect dependent variables and were interpreted by using internal quality attributes. Finally, qualities (Section 7.1) were indirect dependent variables and were interpreted by using factors.

9.9 Extraneous and Confounding Variables and Mitigation of Threats to Validity

The threats to the validity of an experiment refers to anything that can occur during the experiment that makes it difficult or impossible for the researcher to say that the experimental variables caused the changes on the dependent variables and not something else [96][98]. In other words, extraneous and confounding variables were other factors that might compete with dependent variables being used in the interpretation the outcome of the this research, hence, that were threaten the validity of experiment results. Below is a list of the potential extraneous variables in this research experiment along with proposed mitigation strategies to control their effects on the research experiment output.

- *Level of Experience for Developers*: In our experiment, I formed or selected the groups under study, manipulates the treatments for the groups, attempts to control extraneous or confounding variables. In other words, the selection criteria for hiring the participants, and survey to evaluate their skill levels reasonably mitigated its effect.
- *Capacity of work and Task Performance*: There was a reason to believe that the time of days that the experiment conducted, that might affect the capacity of work and concentration of participants.

In our research, training of participants for specialized skills, needed in this experiment, reasonably mitigated the effect of this extraneous variable.

- *Intelligence* and *Anxiety*: No sufficient mitigation strategy to control this threat.
- *Demand characteristics*: Developers got non-verbal interpretations about what is going on in the experiment, i.e., UML and other visual paradigms.
- *Health Factors*: No sufficient mitigation strategy to control this threat.
- *Work Environment:* There was no adequate mitigation strategy to combat this threat, but I prepared the developer's computer to be ready to the implementation, additionally, I prepared an additional computer in the lab to keep a good environment as possible for developers, who were worked in this lab.
- *Personnel Commitment of Developers for Better Design:* I found no sufficient mitigation strategy to control this threat.
- *Accuracy in Manual Measurements*: More than one person participated in measuring some of EQMTA code metrics.
- *Accuracy in Tool's Measurements:* When I used tools meant for measurement, I assumed that they are correct and accurate, however measuring tools are not always right. Therefore, I created a stable environment to apply the automated measurement. I used the Amazon Web Services to create a cloud computing environment [109]. Additionally, I asked human resources to manually calculate some of the measurements using EQMTA metrics, which crosschecked the tool's automatically-generated measurement with manual ones and hence effectively mitigated the inaccuracy risks.

9.10 Restrictions

To compare the EQMTAs' metrics, all the developers must follow the same rules. The technology used to develop the application were: AspectJ and *TransJ* programming languages with Java, JbossAS, JTA, AspectJ, EJB, JPA and Java Servlet. Each participant used the *Eclipse Integrated Development Environment* (IDE) or JBoss Studio with the maven, jboss, junit, log4j and jta1.1 plugins installed. The source code was committed to the BitBucket repository after each completed user activity.

9.11 Measuring Characteristics of the Participants Code Using the Metrics in the EQMTA.

I compared implementations of different sample applications across two study groups: one for *TransJ* and another for AspectJ. The purpose is that the comparison of the two alternative implementations, with a focus on seven secondary quality attributes: understandability, extensibility, localization of design, obliviousness, efficiency, predictability, and scalability. The empirical study was conducted to measure each of these secondary quality attributes by matching a set of metrics that are consistent with the related-internal quality attributes. Each hypothesis that was tested by means of this study can be expressed through the following list of metrics of the EQMTA, We used both manual computation and automated tools to compute measurements for all these metrics (29 metrics), such as tools in [22][72][73].

- We computed the metrics of the first hypothesis, "*Better Encapsulation and Separation of Concern (SoC),* by:
  - CDTA: Calculates the concern occurrences in classes and aspects that can in application-level components. In other words, concern occurrences can be on an aspect or a class that required gathering the context data needed for a transaction-related concern. It is a manual calculation.
  - CDTO: Calculates the total number of transaction operations in an application-level component containing the concern related occurrences, i.e., relevant operation context information. It is a manual calculation.
  - CDLOC: Calculates the total lines of code that implement a given concern in an application-level components. In addition, it calculates the amount of transition points to gather the relevant transaction-related context information by analyzing program code line by line. It is a manual calculation.
    - Predictions: For this hypothesis to hold, we expect that:
      - CDTA, CDTO, and CDLOC will decrease when using *TransJ.*
- We computed the metrics of the second hypothesis, *Supporting a better Loose Coupling,* by:
  - CBC: Counts the total number of direct and indirect interdependencies between components of a transaction program. It is a manual calculation.

  - DIT: Maximum hierarchical distance from component object in the inheritance hierarchy, i.e., from the leave component object to the parent object of the hierarchical tree. It is a manual calculation.
  - CIM: Calculates the total number of classes, aspects, or interfaces explicitly named in the pointcuts of a given aspect. It is a manual calculation.
    - Predictions: For this hypothesis to hold, we expect that:
      - CBC, DIT, and CIM will decrease when using *TransJ.*
- We computed the metrics of the third hypothesis, *Higher-Cohesion and Less Tangling,* by:
  - LCTO: It is a measure for the cohesiveness of a transaction component and is calculated with the *Equation 3,* called Henderson-Sellers method [54]. If (*M(A)*) is the number of methods accessing a transaction attribute *A*, it calculates the average of *M*(*A*) for all attributes, subtracts the number of methods m and divides the result by (1-*M*). A low value indicates a cohesive transaction component and a value close to 1 indicates a lack of cohesion and suggests the component might better be split into a number of subcomponents [30]. The tool [72] calculates this metric.

$$LCTO = \frac{<\rho> - |M(A)|}{1-|M(A)|} \qquad \text{Equation (3)}$$

$where$:

M: be the set of methods defined by the transaction component.

A: be the set of attributes defined by the transaction component.

ρ(*M(A)*) be the number of methods that access attributes *A*, where *a* is a member of *A*.

$< \rho >$ be the mean of $\rho(A)$ over *A*.

    - Predictions: For this hypothesis to hold, we expect that:
      - LCTO will decrease when using *TransJ*.
- We computed the metrics of the fourth hypothesis, *Reducing the software code size* and Complexity, by:

- LOC: Calculates the total executable lines of code with excluding non-executable lines such as white spaces and comments. The tool [72] calculates this metric.
- MLOC: Calculates total executable lines of code for a method or advice, ignoring white spaces and comments. The tool [72] calculates this metric.
- TLOC: Calculates total executable lines of code for a transaction ignoring white spaces and comments. The tool [72] calculates this metric.
- NTO: Calculates the total number of operations in a transaction component. The tool [72] calculates this metric.
- VS: Calculates the total number of transaction components, which include inner classes, aspects, and classes. The tool [72] calculates this metric.
- WOTC: Sums up the McCabe Cyclomatic Complexity for all methods and advices in a transaction component. The tool [72] calculates this metric.
- CC: Calculates the number of flows through a piece of code. Each time a branch occurs (for (...) if (...), switch (...), while (...), for (...), and catch (...) and then ?: ternary operator, as well as the && and || conditional logic operators in expressions) this metric is incremented by one [70]. Calculated for methods/advice only. The tool [72] calculates this metric.
- RFM: Calculates the number of advices and operations possibly accomplished by a given transaction in response to a received request. It is a manual calculation.
  - Predictions: For this hypothesis to hold, we expect that:
    - LOC, MLOC, TLOC, NTO, WOTC, CC and RFM will decrease when using *TransJ.*
    - VS will increase when using *TransJ*.
- We computed the metrics of the fifth hypothesis, *Increasing the capacity of the software obliviousness*, by:
  - NITD: it calculates the number of Inter-Type Declarations (ITD) in the aspects and the number of times they are used, which also includes their references in the application classes. It is a manual calculation.
  - CDA: it calculates the number of modules that are affected by an aspect, which also includes their references in the aspects. It is a manual calculation.

    - ASTC: it calculates the number of distinct application components in the transaction-related concerns, which includes both the distinct number of transaction components and number of transaction operations for those components. It is a manual calculation.
    - ASTO: it calculates the number of methods and advices in the concern containing the references of transaction operations. It is a manual calculation.
  - Predictions: For this hypothesis to hold, we expect that:
    - NITD, CDA, ASTC, and ASTO will decrease when using *TransJ*.
- We computed metrics of the sixth hypothesis , *Improving the software efficiency,* by:
    - MRT: Calculates the rate of the elapsed time from the moment that a user requests a transaction until the time that the transaction server application indicates the request has completed, i.e., committed or aborted. The tool [22] calculates this metric.
    - NCT: Calculates the total number of committed transactions per second. The tool [22] calculates this metric.
    - NUCT: Calculates the total number of aborted (i.e., rolled back) transactions per second. The tool [22] calculates this metric.
    - ToT: Calculate the total number of time-out transactions per second. The tool [22] calculates this metric.
    - RTPM: Calculates the speed of transaction processing. It is a manual calculation.
  - Predictions: For this hypothesis to hold, we expect that:
    - MRT, NUCT, ToT, RTPM, and NCT for *TransJ* will be equal MRT, NUCT, ToT, RTPM, and NCT for AspectJ. Otherwise
      - MRT, NUCT, and ToT will decrease when using *TransJ*,
      - RTPM, and NCT will increase when using *TransJ*.
- We computed the metrics of the seventh hypothesis , *Improving extensibility,* by:
    - NoCA: Calculates the number of changes required to reuse the concern for another application. The eclipse IDE calculates this metric.

  - NoC: Calculates the number of changes required to maintain the concern. The eclipse IDE calculates this metric.
    - Predictions: For this hypothesis to hold, we expect that:
      - NoC and NoCA will decrease when using *TransJ*.

- We computed metrics of the eighth hypothesis, *Improving the productivity of the development process,* using [102]:
  - AT: Calculates the total time during which programmers are writing the required code to maintain the concerns. It is a manual calculation by the questionnaire.
  - PT: Calculates the total time during which programmers are reading, understanding the code design, and fixing bugs. It is a manual calculation by the questionnaire.
  - NoB: Calculates the total number of bugs during the system development to reuse and maintain the concerns. It is a manual calculation through the questionnaire.
    - Predictions: For this hypothesis to hold, we expect that:
      - AT, PT, and NoB will decrease when using *TransJ*.

CHAPTER 10

EXPERIMENT RESULTS AND INTERPRETATIONS

This chapter presents empirical results relevant to the eight hypotheses. We analyzed and evaluated the reusability and performance using various software artifacts, which included surveys, questionnaires, hourly journals, maintenance history, and actual code. The collected data are presented using column bar graphs, allowing for inspecting of the contributions of the application base code, base aspects, reusable aspects, and other aspects to each metric in each programming activity.

The number of participants who have taken part in our study is limited. Since internal quality metrics are computed for each method, advice, class and aspect in the software system, the number of data appeared sufficient for executing the reliable preliminary experiment to measure impact of *TransJ* and Aspect on design quality metrics. In the following graphs, the vertical axes represent the measurements, and the horizontal axes represent the three activities of the experiment. For each activity there are two bars: a blue bar for the results of AspectJ group and an orange bar for the results of *TransJ* group.

10.1 Better Encapsulation and Separation of Concern (SoC)

The first hypothesis theorized that if transaction-related crosscutting concerns are effectively encapsulated in *TransJ* aspects, the software has a better separation of concerns and less scattering as measured by CDTA, CDTO, and CDLOC than equivalent systems developed with AOP design techniques. In other words, the CDA, CDO and CDLOC metric values for *TransJ* should be less than AspectJ as described in Section 9.11.

From the graphs in Figure 10-1, we found that the interest average of CDA, CDO and CDLOC values for *TransJ* went to zero in all three activities of the experiment, and the result was significantly different from AspectJ in the all activities. This was caused by: the parameterization of advice via joinpoints, which enables aspects to be customized in different contexts and thus increase the reusability of aspects, resulting in the elimination of application-specific code from the transaction implementations; and the *TransJ* pointcuts provide total localization and obliviousness between the transaction application and transaction-

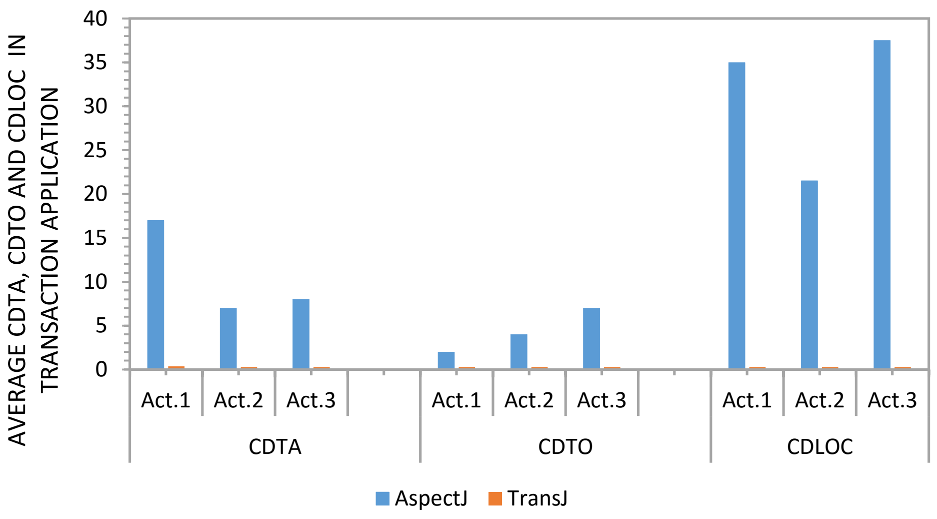


Figure 10-1. CDTA, CDTO, and CDLOC Coverage over Activities

related crosscutting concern. Obliviousness can thus be handled for increased SoCs within the reusable aspect implementation for *TransJ* pointcuts. This technique can be seen as a compromise, where the reusable aspect can completely encapsulate the transaction-related crosscutting concern from the core code.

In AspectJ, components and their operations for crosscutting concern were significantly more diffused in the transaction application because the pointcuts had to be tied to programming constructs instead of transaction abstractions. Thus, *TransJ* provides a mechanism for encapsulating crosscutting concerns that apply to transactions into modular units; this mechanism provides an easy approach to identify the joinpoint where only the existence of the transaction-related concern and its associated contextual requirements for perfection of reusability.

From these results, they are apparent that the first hypothesis hold true for better encapsulation, localization, obliviousness and separation of concerns with no scattering in *TransJ* implementations than in AspectJ.

10.2 Supported a better Loose Coupling

The second hypothesis theorized that if the transaction-related crosscutting concerns are effectively encapsulated in *TransJ* aspects, then the software has a lower coupling (as measured by CBC, DIT and CIM) than equivalent systems with AOP design techniques. Specifically, the values of CBC, DIT and CIM for *TransJ* should be less than AspectJ. Figure 10-2 shows that *TransJ* implementation decreased the values of these metrics, as compared to *AspectJ* implementations in all the three activities of the experiment. *TransJ* removed dependencies and did not maintain any direct relationship between transaction-related crosscutting concerns and the core application components. Removing dependencies between the DTPS components improved the DTPS architecture, which lead to a less coupled architecture, i.e., lower CBC value, with less complicated design easier to modify or adapt to new requirements, additionally, it increased the velocity of adding new features (see Sections 10.7 and 10.8).

In AspectJ, unnecessary coupling of transaction-related concerns with the core application components increased CBC, which hindered reuse and code understandability. The reason for the tight coupling in AspectJ compared with *TransJ* was that the complexity inherent in precisely describing sets of joinpoints when these joinpoints have no explicit name, this makes them difficult to define pointcuts, such that they anchor transaction crosscutting logic precisely, where needed without unintentionally matching additional joinpoints. This means that joinpoints become more likely for the semantics of a pointcut to changes as the core code evolves, i.e., a fragile pointcut. Whilst, *TransJ* defines a set of elegant and clear joinpoints and a set of reusable stable pointcuts.

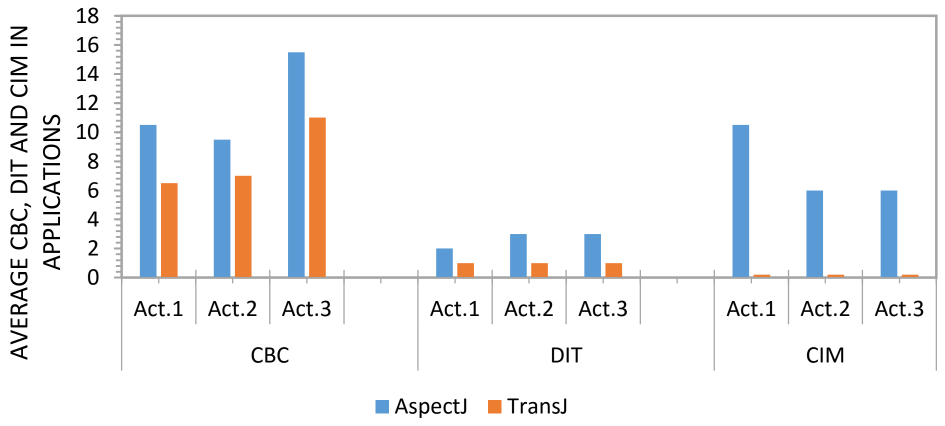


Figure 10-2. CBC, DIT and CIM Coverage over Activities

On the one hand, wide variations were found in DIT and CIM metrics from *TransJ* group and AspectJ group. The most significant indicator of the decrease in coupling between aspects and the core code is the impact of *TransJ*'s joinpoints on the CIM metric. This metric counts the number of modules explicitly named in pointcuts. Compared to the AspectJ activities, the *TransJ* activities have a reduction of 100%, 100% and 100% in CIM (i.e., all of the three activities have an average value of zero for CIM metric). This was caused by providing a comprehensive set of pointcuts, which fully encapsulates the distributed transaction abstractions. This allows participant programmers to reuse the pointcuts directly, so they did not need to override or inherit the aspect components to name in the pointcuts of a given class. In contrast, AspectJ programmers suffered from a lack of clarity of relationship among transaction-related concerns and application components, wherein aspects acquire context information from one of more classes. Thus, they preferred to inherit all of the attributes and operations from parent (superclass) methods in crosscutting concerns to share context data across aspects and distributed transaction application components.

In consonance with these results, we present additional insights into the effects of *TransJ* on comprehensibility, readability, and reusability. In other words, *TransJ* provides better understanding the overarching among obliviousness and reuse. Thus, we can confidently conclude that the second hypothesis hold true for reduced coupling in *TransJ* compared with AspectJ, therefore, increase the understanding and reuse of the DTPS components.

10.3 Improved Cohesion and Less Tangling

The third hypothesis theorized that if transaction-related crosscutting concerns are effectively encapsulated in *TransJ* aspects, then the software has a higher cohesion (See Section 9.11) and less tangling than equivalent systems developed with AOP design techniques. In other words, the values of LCTO metric for *TransJ* should be less than AspectJ. In Figure 10-3, the result reveals that *TransJ* maintains a lower value for LCTO than AspectJ in all the three activities of the experiment. Thus, *TransJ* promoted encapsulation with implementing a more independent component that implements a single logical function (more cohesive) than implemented with AspectJ.

Compared to the AspectJ group, the *TransJ* group improved cohesion in all activities, sometimes significantly (from 8% to 75%). The decrease in the cohesion of the AspectJ activities is caused by the need

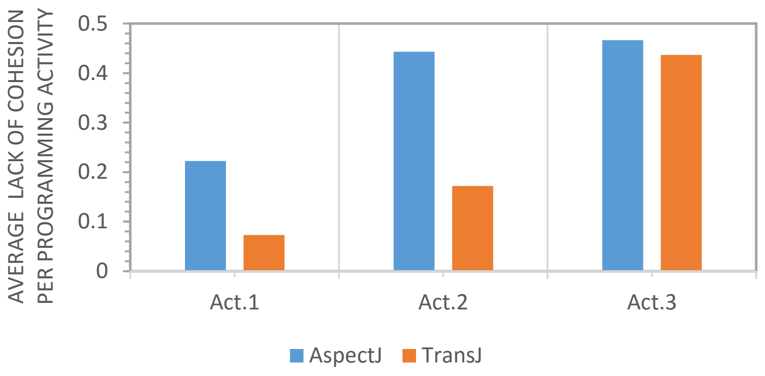


Figure 10-3. LCTO Coverage over Activities

to extract new methods to expose advisable joinpoints i.e., multiple transaction joinpoints cannot be advised as an atomic unit (e.g., begin – commit, begin – abort or lock – release). In other words, the entire transaction cannot be advised as execution unit, which results in the transaction operation refactorings that decrease cohesion, i.e., increase the LCTO value. Careful inspection of these results shows that in all activities there are improvements in cohesion in the *TransJ* activities versus the AspectJ activities. From these results, we proved that the third hypothesis hold true for increased cohesion in *TransJ* compared with AspectJ.

10.4 Reduced the Software Code Size and Complexity

The fourth hypothesis theorized that if transaction-related crosscutting concerns are effectively encapsulated in *TransJ* aspects, then the software is less complex and not significantly larger than the equivalent systems developed with AOP design techniques (as described by LOC, MLOC, TLOC, NTO, WOTC, CC, RFM and VS in Section 9.11). In other words, the values of LOC, MLOC, TLOC, NTO, WOTC, CC and RFM metrics for *TransJ* should be less than AspectJ and VS for *TransJ* should be more than AspectJ.

Figures 10-4 through 10-8 show that *TransJ* implementations decreased the metric values for LOC, MLOC, TLOC, NOT, WOTC, CC and RFM and increased VS value in all the three activities of the experiment.

In comparison with *TransJ*, AspectJ programmers found the aspects and application code tends to contain very terse pointcuts, advices and extra code, especially, when combined with transaction constructs,

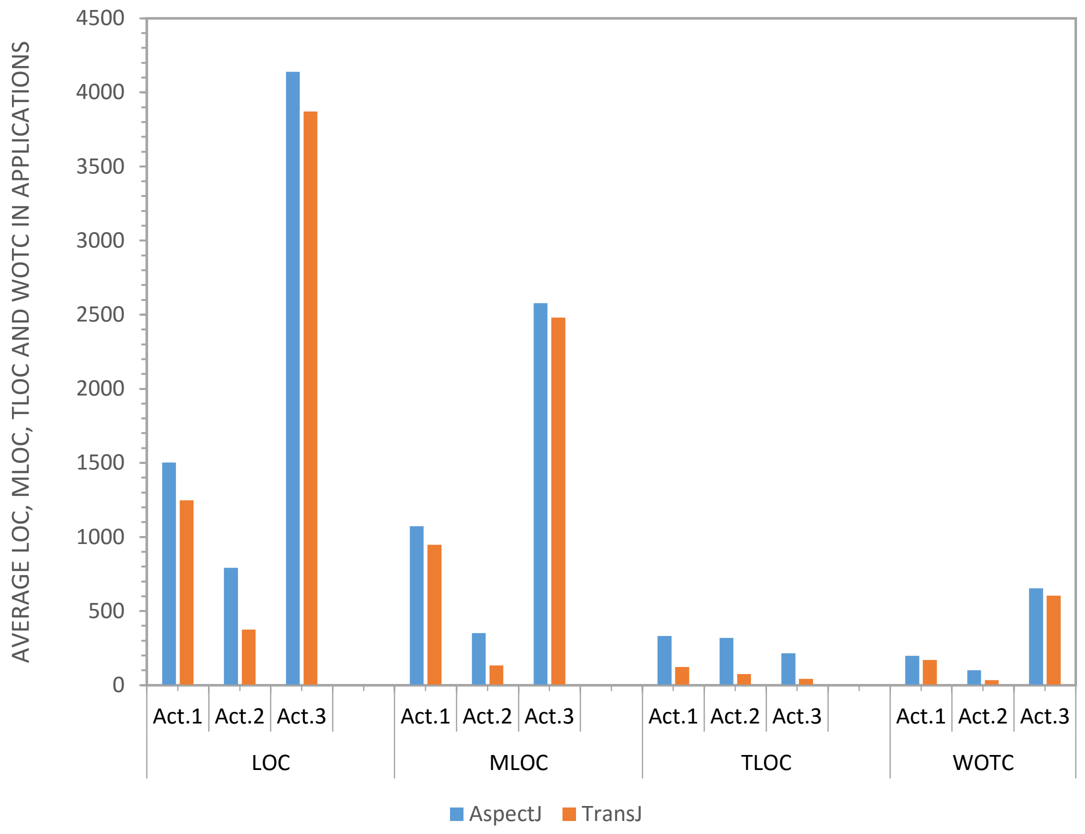


Figure 10-4. Average LOC, MLOC, TLOC and WOTC over Activities

such as transaction demarcations, to pull all relevant context information. This had affected on the capacity of creating and understanding, modifying, reusing, and maintaining applications. In *TransJ*, programmers found the aspect code more elegant, classy and clean set of pointcuts in transaction abstractions, which helped them to code the crosscutting concerns in less LOC compared with AspectJ. Particularly, the MLOC and TLOC count the total lines of code that implement the transaction-related crosscutting concern component and transaction operations, including *TransJ*'s joinpoints, which referred to the context of the core code. The *TransJ* group performed significantly better than AspectJ group for all activities (by 32% and 80%), as shown in Figure 10-5. In *TransJ*, two induced factors affect these metrics: the UMJDT model captures various general distributed transaction abstractions in meaningful, reusable joinpoints and a set of base aspects, which help developers implement the transaction-related crosscutting concerns in simpler and logical method

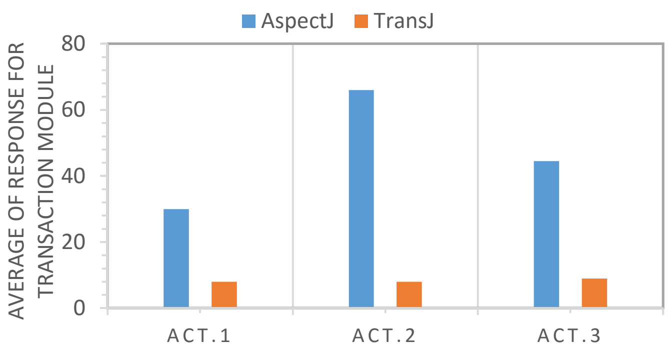


Figure 10-5. Average RFM over Activities

bodies, i.e., advice, with no extra lines of codes and less number of operations and advices, thus this reduced the RFM value. Second, *TransJ*'s joinpoints referenced by broad contexts and stable pointcut definitions, therefore, applications did not need additional context information, such as an identifier or lock snapshot. This allowed the reusable and application-level aspects to inherit or reuse pointcuts to apply the logic of transaction-related crosscutting concern in appropriate transaction places. Hence, *TransJ* reduced the values of MLOC, TLOC, NTO, WOTC, and RFM. Consequently, it reduced the bug density (see Section 10.8 for more details).

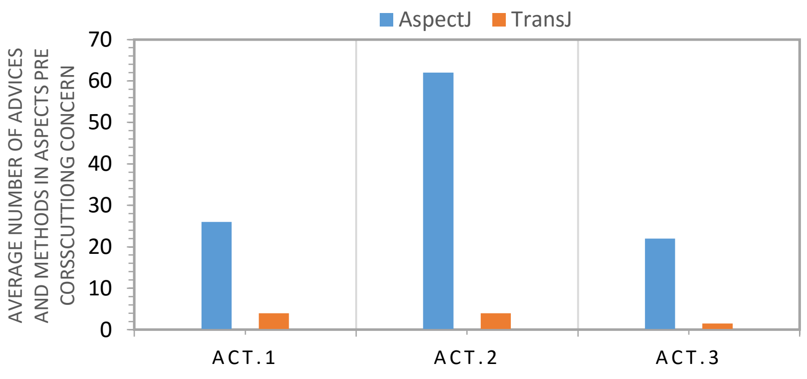


Figure 10-6. Average NTO over Activities

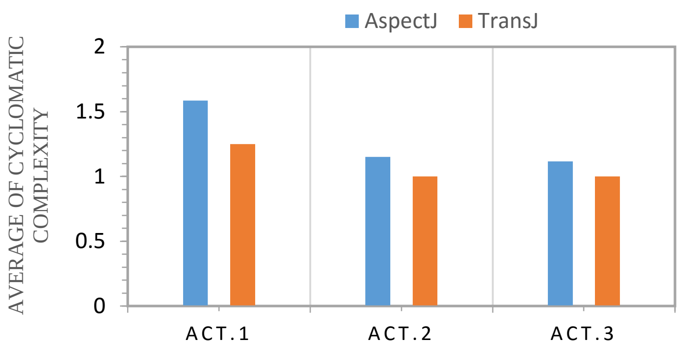


Figure 10-7. Average CC over Activities

Figure 10-7 shows that the value of CC is smaller for *TransJ* than AspectJ, because *TransJ* hides complex transaction abstractions, as mentioned, which result in simple conditional statements and less tangled code.

As predicted by the above hypothesis, results shown in Figure 10-8 give sufficient evidence that the average VS value of all programs was more for *TransJ* than AspectJ, due to inlined code in transaction scopes being extracted and gathered to inner classes, i.e., contexts and base aspects (caused improvements of 12% to 23%). Although the number of components were more in *TransJ* implementations, but they were more

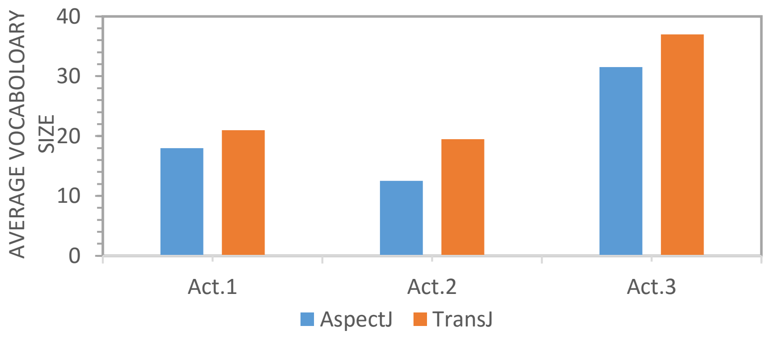


Figure 10-8. Average VS over Activities

cohesive. From these results, we can confidently conclude that the fourth hypothesis hold true for less complex and a small code size software in *TransJ* compared with AspectJ.

10.5 Increased the capacity of the software obliviousness

The fifth hypothesis theorized that if transaction-related crosscutting concerns are effectively encapsulated in *TransJ* aspects, then the software will be more oblivious than equivalent systems developed with AOP design techniques (as described by NITD, CDA, ASTC and ASTO in Section 9.11). The values of NITD, CDA, ASTC and ASTO for *TransJ* should be less than AspectJ. Figure 10-9 shows that *TransJ* implementations significantly reduced the values of NITD, CDA, ASTC and ASTO metrics.

Compared to the AspectJ, NITD and CDA for all *TransJ* activities differed by 100%. The reason for having this result, i.e., zero value, for NITD and CDA in *TransJ* activities that the programmer avoided the unnecessary complexity of the selected applications, due to *TransJ*'s joinpoints parameterization that facilitated generic transaction-related crosscutting concern logic, which is free from application couplings, and also avoided couplings caused by the application-level advice. In other words, *TransJ* programmers used

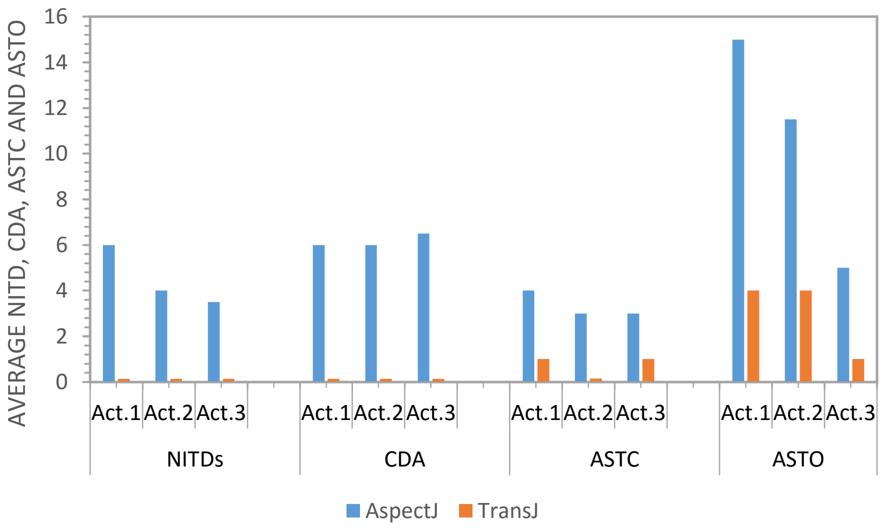


Figure 10-9. Average NITD, CDA, ASTC and ASTO over Activities

transaction abstractions and did not need to use *Inter-Type Declarations* (ITDs) for sharing of context information between application and aspect components.

The ASTC and ASTO assess the scattering of an aspect using different level of transaction granularity, i.e., transaction and transaction operations, respectively. Significant reduction in ASTC and ASTO was due to the layers of indirections among the transaction application and aspect components, which *TransJ* provides but are missing in AspectJ. In other words, *TransJ*'s base and/or reusable aspects allow application-level code to customize the *TransJ*'s joinpoints without coupling links, due to remove transaction-related concept logic. This decreases the ASTC and ASTO values in the core code, e.g., in activity 1 the reduction was around 80%.

In a nutshell, The improvement of the *TransJ* activities verse the AspectJ activities was caused by (a) the higher level of reuse of base aspects, and (b) scoped joinpoints, i.e., contexts, eliminating the need to create operations to expose new joinpoints. This again underscores the benefits of improving obliviousness where pointcuts allow reuse in an oblivious approach. From these results, we can confidently conclude that the fifth hypothesis hold true for less oblivious software crosscutting concerns in *TransJ* compared with AspectJ.

10.6 Preserved the software efficiency

To evaluate this hypothesis, we conducted the experiment to assess performance of all activities within a stable environment to reduce noise. We launched two instances using the *Amazon Web Service*

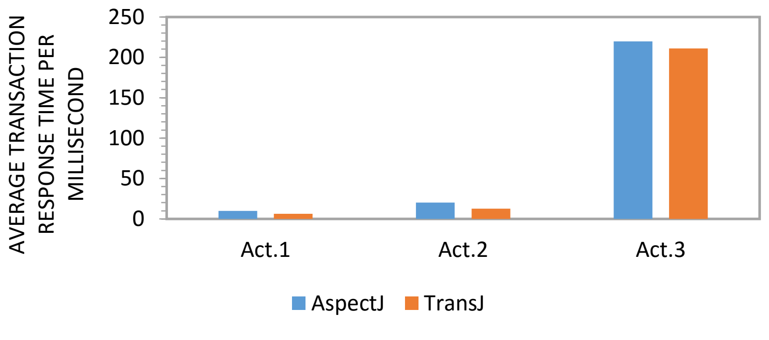


Figure 10-10. Average MRT over Activities

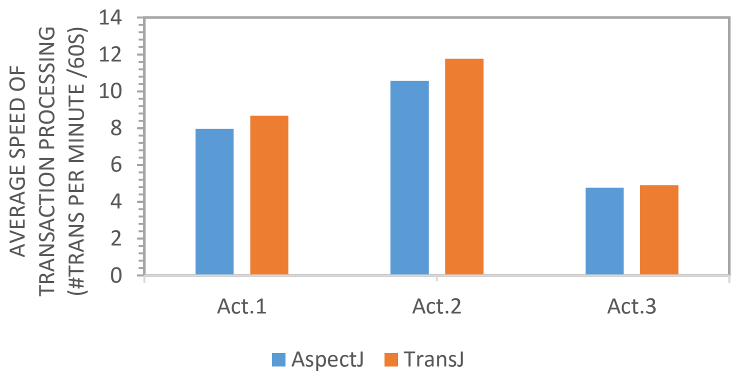


Figure 10-11. Average Transaction Velocity (RTPM) over Activities

(AWS) Management Console for executing applications with a little noise: a Windows instance and an Amazon Rational Database Service (Amazon RDS) instance [109]. They are virtual servers in the AWS cloud. First, we set up and configured a windows instance within a type of t2.small, which provided the 64-bit version of Microsoft Windows Server 2008 R2, 40G hard drive, 2G memory (RAM). Then we configured the application to run on this instance, with JDK7, JBoss server, Maven, Eclipse, JTA, and *TransJ*. Then we deployed a MySql database on the RDS instance.

The sixth hypothesis theorized that if transaction-related crosscutting concerns are effectively encapsulated in *TransJ* aspects, then the software would preserve or improve runtime performance compared with equivalent systems developed with AOP design techniques (as described by MRT, NCT, NUCT, RTPM and ToT in section 9.10). The values of MRT, NUCT and ToT for *TransJ* should be equal or less than AspectJ, and RTPM, NCT for *TransJ* should be equal or more than AspectJ.

Figures 10-10 through 10-12 show that *TransJ* implementation slightly decreased the metric values for MRT, NUCT, ToT, and slightly increased NCT with maintaining the RTPM in all three activities of the experiment.

TransJ allows dynamic weaving of aspects at run-time by looking up to the contexts instead of needing to programing by hand as is done in AspectJ. This impacts the implementation of runtime weaver at several points. At first, it instantiates new context information into the class at the right time (just-in-time) for gathering the data needed to perform the behavior specified by the aspect. Second, it wraps all transaction

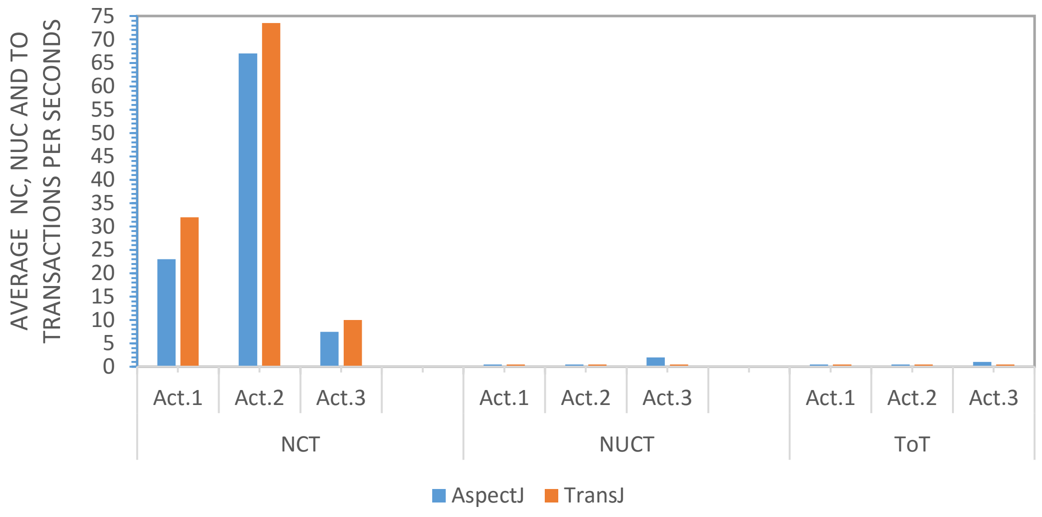


Figure 10-12. Average NCT, NUCT and ToT over Activities

operations by demarcating the original methods and then replacing the core implementation with the specification of the original one, such as name, argument, returned data, etc., which contains the wrapping code in its body. This wrapping can lead to efficient method invocations.

Figure 10-10 and Figure 10-11 indicate that the *TransJ* group performed very slightly better than the AspectJ group for Act.1 and Act.2 with almost 0% improvement for Act.3. This lack of improvement for Act.3 was caused by the overhead of creating a transaction and transaction operation thread instances, synchronization and the high concurrent potential for conflicts over the shared resource. In other words, there are no major differences between the efficiency of *TransJ* activities and AspectJ activities.

Figure 10-12 shows that the results for the NUCT and ToT metrics remained the same for the Act.1 and Act.2. However, in Act.3 *TransJ* decreased very slightly the potential of having better ToT and NUCT values. The decrease in NUCT and ToT values in *TransJ* at Act.3 was caused by exposing advisable joinpoints, i.e., lockingJP and resourceLockedJP and dynamic weaving of aspects on them. These joinpoints represented an indication of the benefits that can come when concurrent operations access the shared resource. However, there are no major differences between the throughput of *TransJ* activities and AspectJ activities.

In a nutshell, the results of figures do not give sufficient evidence to claim that the benefits of improving software performance. But from these results, we can confidently conclude that the sixth hypothesis hold true: preserving runtime performance in *TransJ* compared to AspectJ.

10.7 Improved reuse and software extensibility

The seventh hypothesis theorized that if transaction-related crosscutting concerns are effectively encapsulated in *TransJ*, the crosscutting concern would require a smaller number of changes to reuse compared to equivalent systems developed with AOP design techniques (as measured by NoC, NoCA in Section 9.10). In other words, NoCA and NoC values for *TransJ* should be less than AspectJ. From the results shown in Figure 10-13, we can see that *TransJ* implementation significantly reduced the changes required to reuse the performance measurement concern implementations in Act.1 and Act.2 of the experiment compared to AspectJ. This means that the application is more amenable to extension.

Compared with AspectJ, the presence of joinpoints in the base aspect of *TransJ* allows the implementation of the crosscutting concern logic in reusable and application-level aspects, which allow contexts and crosscutting concerns to be explicitly communicated. Figure 10-13 presents the percentage of crosscutting concerns that were implemented by abstract aspects (in base aspect). The data confirm that

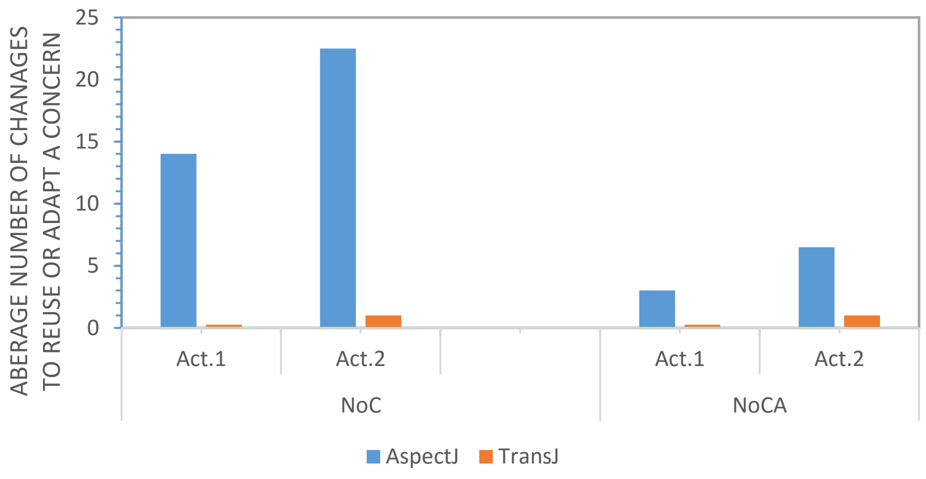


Figure 10-13. Average Number of Changes of Performance Measurement Concern over Conference Registration System and Bank System Activities

significant increases in reusability can be gained by applying *TransJ*'s joinpoints where appropriate. In other words, *TransJ* aspects were overall more oblivious, localizing, logical and independent from the base application than AspectJ concerns, thereby reducing the NoC value in all activities of the experiment.

Figure 10-13 shows the number of changes required to reuse and adapt the concern and application in Act.1 compared to Act.2, was significantly less for *TransJ* than AspectJ. The reason for the difference between NoC and NoCA metrics is that in NoC we are only considering changes in the concern; whereas in NoCA, we are interested in the number of changes both in the concern and application. We found that *TransJ* concerns still were overall more oblivious, localizing, logical and independent from the base application than AspectJ concerns, and so they have reduced NoCA values in Act.1 and Act.2 of the experiment.

Figure 10-14 provides another graphical representation of the analysis of reuse for AspectJ and *TransJ*. The orange-colored graphs represent scattering in *TransJ* (aspects only) and the blue-colored graphs represent scattering in AspectJ implementations. The scattered points in the graph indicate that the number of changes required for reusing a concern with *TransJ* and AspectJ in different activities, respectively. The scattered points represent ASTC, ASTO, CDA, NITD, CDTA, CDTO, and CDLOC metrics results. Overall, the results of the graph indicate that the CDA, NITD, CDTA, CDTO, and CDLOC remained zero for the

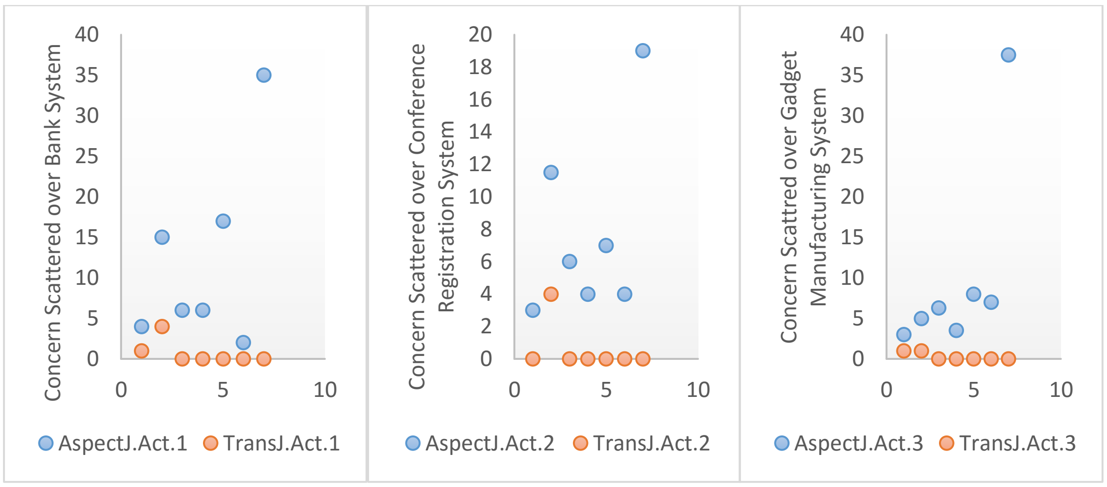


Figure 10-14. ASTC, ASTO, CDA, NITD, CDTA, CDTO and CDLOC over Activities of AspectJ and *TransJ*

activities of *TransJ* (highly reusable and more extensible), but were highly scattered for AspectJ. The reason for less scattering is discussed in Section 10.1 and 10.5 above.

From these results, we can confidently conclude that the seventh hypothesis hold true: more reusability and extensibility in *TransJ* compared to AspectJ.

10.8 Reduced the time of the development process

The eighth hypothesis theorized that if *TransJ* provides a better modularization of transaction-related crosscutting concerns, then the development transaction system would be less complicated, and more predictable and readable. Thus, software development efficiency (measured by AT, PT and NoB) would be increased, so that the software would be created faster than the equivalent software developed with AOP design techniques. In other words, AT, PT and NoB values for *TransJ* should be less than AspectJ.

From the results shown in Figures 10-15, we can see that *TransJ* significantly reduced the period that required to read, understand, implement, and debug the implementations of transaction-related crosscutting concerns in all activities of the experiment compared to AspectJ. These results confirm that the applications were more flexible to implement with *TransJ* and were robust with respect to bugs and error compared to the AspectJ implementation (Section 10.9 provides more details.)

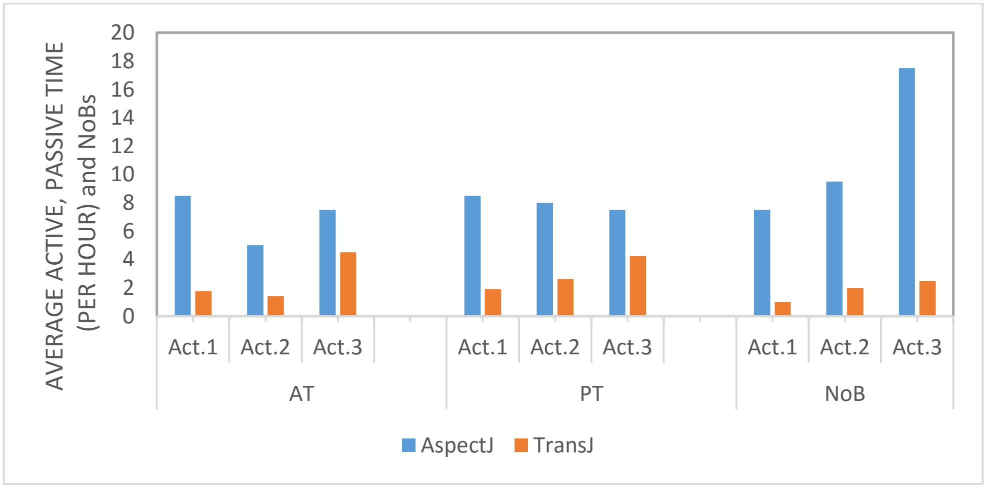


Figure 10-15. Average AT, PT and NoBs over Activities

In addition, these figures indicate that *TransJ* implementation decreased the values of these metrics, as compared to *AspectJ* implementations in all the three activities of the experiment. We found that the average AT that *TransJ* participants spent on typing and producing actual code of the transaction-related crosscutting concern was less than for AspectJ participants. It is interesting that the developer of all three activities needed approximately the same amount of AT to finish the activity. Related to this hypothesis, the first hypothesis claimed that the *TransJ* participants, compared to AspectJ participants, found the *TransJ* activities were well-separated concerns, so that the total developer's working time was shorter than the development time of the equivalent system developed with AspectJ. Therefore, the *TransJ* participants performed significantly better than the AspectJ participants for all activities.

PT represents the amount of time they spent on reading the source code, understanding secondary requirements and looking for bugs. The increases in the PT in the AspectJ activities are caused by the need to study the whole code to find new pointcuts to expose advisable joinpoints and to gather the relevant information to a specific context that is required to weave the crosscutting concerns of appropriate joinpoints.

In contrast, *TransJ* provides pointcuts that help developers code the crosscutting concerns obliviously. In addition, they do not need to create shared data structures, i.e., contexts, to have an explicit cooperation between base application code and aspects. This one simple benefit in the mindset of programmers can drastically reduce the number and seriousness of bugs, i.e., NoBs. NoB counts the number of bugs and errors raised to maintain the crosscutting concerns.

This again emphasizes the benefits of improving software development efficiency. From these results, we can confidently conclude that the eighth hypothesis hold true: less software development time is required for *TransJ* than for AspectJ.

10.9 Other Useful Observations from Questionnaires and Daily Journals

Besides the analysis of the hypotheses via the metrics, we gathered additional quantitative data handful observations from developers' questionnaires (Appendices C and D) and daily maintenance journals during each activity of the experiment.

In regards to understandable code, we found that 100% of AspectJ participants in the Act.1 were confused in identifying pointcuts for implementing the given extension feature, and 50% of the same participants were still confused during Act.2 and Act.3. On the other hand, none of the *TransJ* participants struggled with identifying pointcuts during either phase. This tells us that *TransJ* implementation provides simple pointcuts with understandable distributed transaction abstractions. Figure 10-16 show more percentages of most consuming task during the implementations.

For reusability, we observed that 100% of the AspectJ participants in Act.1, Act2 and Act.3 agreed that their applications might require redesigning the application's structure, making minor changes in the classes, their relationships, roles, and responsibilities in the original application. On the other hand, none of the *TransJ* participants made this observation for any activity and they said that they reused the existing code to implement new changes without failures in all activities. This indirectly reemphasizes the seventh hypothesis, which states that *TransJ* implementations help in developing more reusable crosscutting concerns.

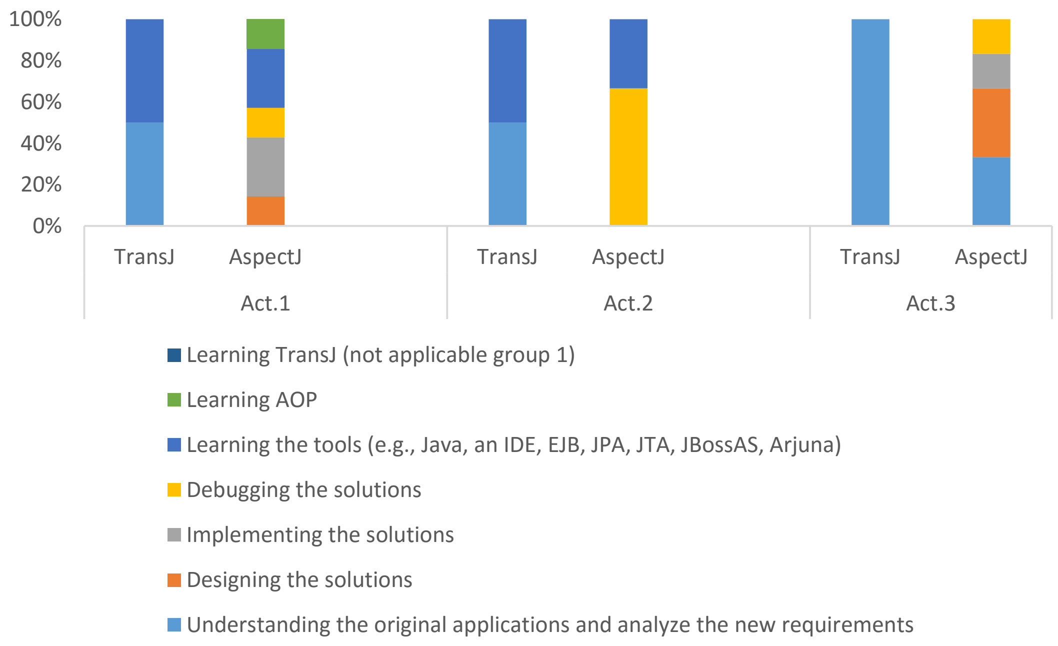


Figure 10-16. The most Time Consuming Activities during Implementation of Feature Changes

Similarly, for extensibility, 100% of the AspectJ participants said that their changes introduced new dependencies and increased LOC and complexity in the original sample application in all activities. However, none of the *TransJ* participants felt that they introduced any dependencies during any activity. Hence, this reemphasizes the seventh hypothesis, which asserts that *TransJ* implementation helps in developing more extensible applications.

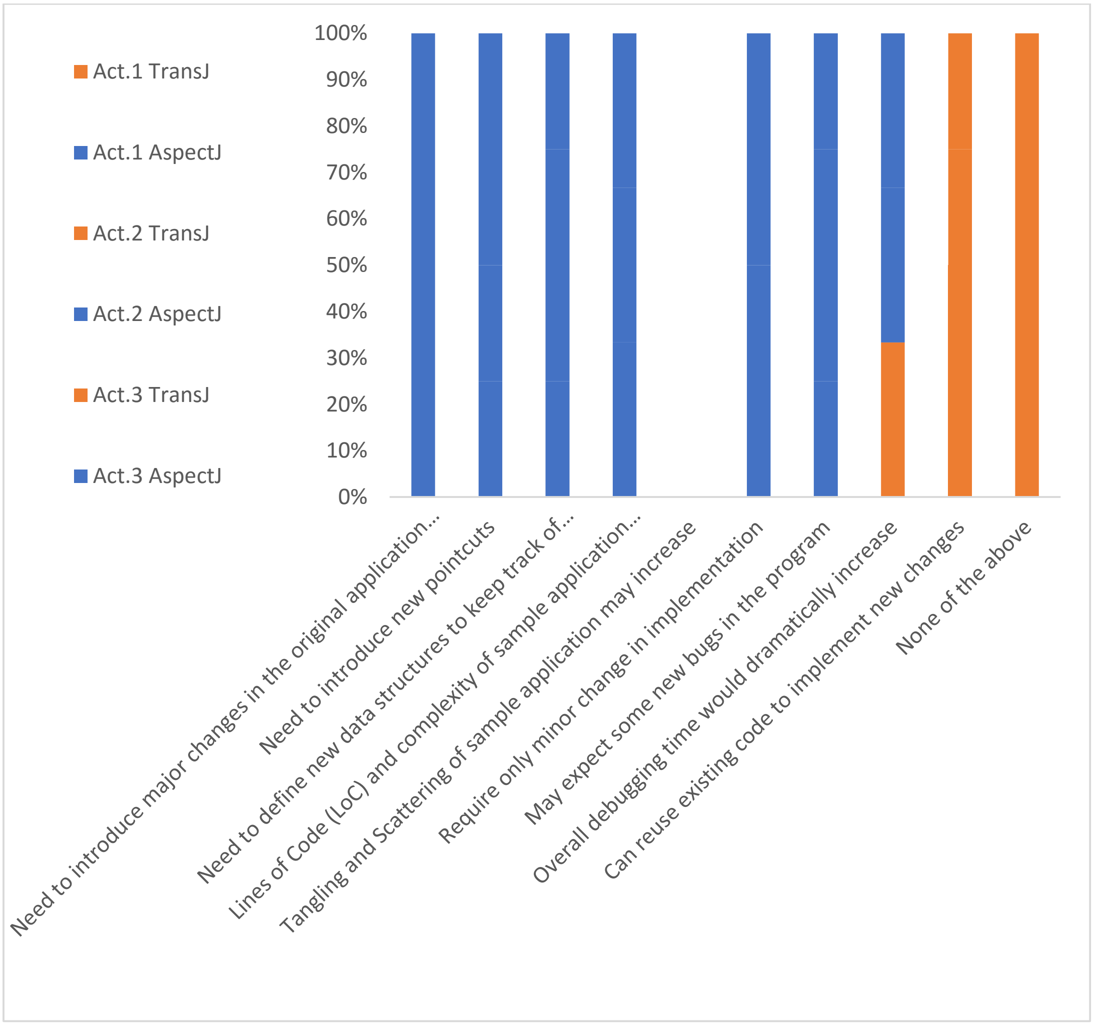


Figure 10-17. Sample of Observation of Changes made in Original Applications by the AspectJ Participants versus the *TransJ* Participants for all Activities

The questionnaires and the maintenance journal history also provide information on the number and frequency of bugs and errors. Specifically, 100% of the participants in the AspectJ group said that their extensions introduced new failures, i.e., bugs, into the application code during activities (see Figure 10-17). The number of failures further increased for Act.2 and increased again with Act.3. However, none of the *TransJ* participants in Act.1 and Act.2 made this statement, while 50% participant stated in the daily journal and maintenance history that they introduced very few bugs, but the total time for developing the concern is decreased compared to AspectJ participants. This tells us that *TransJ*'s modularization and obliviousness decreased the failures and debugging time. Hence, this reemphasizes the eighth hypothesis, which asserts that *TransJ* implementation helps in reducing development time.

For performance, we observed that 100% of the AspectJ participants in Act.1, Act2 and Act.3 ranked their application performance after changes to the original application is fair for all activities. On the other hand, 100% the *TransJ* participants described the overall application a good performance after changes to the original application. This indirectly reemphasizes the hypotheses, which state that the *TransJ* helps in improving the reusability without sacrificing the software efficiency compared to AspectJ.

CHAPTER 11

RELATED WORK

For years, software engineers have strived with the question of how to construct and organize their code with the intention of maximizing flexibility, extensibility, understandability, maintainability, performance, reuse and other software qualities. While OOP provides a solid-framework for code-based manipulation, optimization, organization, it breaks down when programmers must implement features that cut across the entire system, such as logging, distribution, concurrency (synchronization), security auditing, multithreaded, and transaction handling [77]. Implementing such crosscutting functionality do not fit efficiently into OOP, which leads to an unnecessary code duplication, a complex code, a decrease in software quality, and an increase in product errors and bugs [41]. AOP adds a high level model of reuse in traditional OOP frameworks, with minimal impact on existing code bases [3][5].

The ideas, concepts and approaches investigated in this section intersect with a broad spectrum of research projects on distributed transaction processing systems and aspects, reusable AO frameworks, application-level aspects and interactions. We offer some important and relevant research in this chapter.

11.1 Transaction Middlewares and Frameworks

Several research works are currently underway to explore the feasibility of AOP mechanisms to deal with the concepts of transaction in various scenarios, such as [60]. The case study proposed in [77], promises to be a perfect candidate that may serve as a benchmark for evaluating the new AOP approaches, the expressivity of AOP languages, the performance of AOP environments, and the suitability of AO modeling notations. This research presents a language independent decomposition of the Atomicity, Consistency, Isolation and Durability (ACID) properties of transactions into a set of fine-grained aspects, i.e., base aspects, each one providing a well-defined reusable functionality. It then shows how these aspects can be configured and composed in different ways to achieve various concurrency control and recovery strategies for the transactional object. Therefore, this framework enables the design of various concurrency control and recovery concerns through the configuration and composition of these new aspects. However, other concerns, such as transaction life-cycle management, are only primarily supported and badly modularized, and as a result, their functionality cuts through the code design of the other aspects of the framework. Motivated by these, Kienzle was taken the case study one step further. In [49][50], he presents a language-independent framework that provides the runtime support for transactions, called *AspectOptima*. It uses AO technology to decompose transaction models and their implementations into many individual reusable aspects. In other words, it consists of a collection of ten base aspects that can be configured to guarantee the ACID properties for the transactional object [50]. However, this purpose of this research is not produced implementations for a specific transaction standard like JTA, or a reference implementations.

In comparing with AspectOptima, the *TransJ* discusses the composition of transaction abstractions by separating out the definition of transactions from the definition of other aspects using general-purpose abstract transaction concepts, i.e., high-level abstractions, each one providing a reusable functionality. We believe our work enables better reuse, encapsulation and obliviousness for transaction-related crosscutting concerns.

In [78], Panos introduces the transaction framework, called *ACTA*, that provides the formal reasoning about the attributes of transaction notations. It is not a model for transactions, but instead can be used to determine the new transaction patterns to define the effects of transactions on other transactions, and the effects of transactions on other concepts. However, the framework's power comes from its ability to represent the structural behavior of transactions and the relations between them. ACTA uses mainly to capture transaction properties, such as consistency, visibility, permanence and recovery. However, implementations of ACTA models are not always direct and simple. ACTA is a purely meta-model, and therefore does not concern itself with possible implementation issues. Consequently, it is really hard to determine that the ACTA model does actually identify the semantics of a transactional model [78]. On the other hand, *TransJ* serves for a more practical purpose. *TransJ* tries to define a transaction joinpoint model, which is not the only contribution of this research. In addition, it proposes a new context concept to act as meta-data model for encapsulating the transaction-related information. Furthermore, the research adds new abstract concepts, which correspond to the transaction primitives in JTA and AspectJ frameworks, enable the design of various application-level aspects through the configuration and composition of reusable and base aspects.

11.1.1 *JTA Platforms*

Spring framework provides an abstraction for transaction management that presents a programming model for running an application with any JTA implementation [46][47]. It supports the programmatic and declarative transaction management paradigms [44]. The central implementation of Spring framewrork's transaction management is a *JtaTransactionManager*, which is the standard choice to run on J2EE application servers. Spring's transaction applications can use the *PlatformTransactionManager* implementation for JTA, delegating to a backend JTA provider [11]. Moreover, Spring provides several vendor-specific platform transaction managers that are recommended to be used if appropriate, such as *WebLogicJtaTransactionManager* on Oracle WebLogic server, and *WebSphereUowTransactionManager* on IBM WebSphere application server. For J2EE servers, the standard *JtaTransactionManager* is sufficient [45][46].

Atomikos EssentialTransactions [23] is a free open source TM is available, it allows developers to run distributed transactions on a lightweight run-time environment. It provides a fully functional JTA and offers more functions than defined in the JTA specifications [23]. *Bitronix Transaction Manager* (BTM) is similar to Atomikos, but it aims to make JTA a commodity. BTM offers all services required by the JTA API while trying to preserve the code as simple as possible for easier understanding of the XA standard [24].

Narayana is the primer open source TM. It provides a transaction toolkit which allows application developers to create transaction applications using standards-based protocols for all resources, message queues, and other components. Narayana is formerly known as *JBOSS Transaction Services Suite* 5 (JbossTS 5) [47]. JbossTS includes technology acquired in 2005 from the *Arjuna Transaction Service Suite* (ArjunaTS) and *Hewlett-Packard* (HP) technologies, which support both JTA and JTS APIs. ArjunaTS is made up of a number of different modules: ORB Portability layer, SOAP Portability, the *Object Transaction Service* (OTS), ArjunaCore, *Java Database Connectivity* (JDBC), JTA, Transaction Objects, *Web Services transaction support* (WS-T). See [47] for more information. ArjunaCore is a central module that provides the core of all of the transaction service implementations at the heart of all of transaction products. It supports ACID properties as well as nested transactions. JbossJTA allows application programmers to construct both local and distributed JTA transactions, which wraps ArjunaCore to provide local JTA support without distributions, and wraps the JTS to provide distributed JTA transactions. The transaction module provides

OO classes from which application classes can inherit to obtain preferred properties, such as persistence (e.g., StateManager object) and concurrency control (e.g., LockManager object) [21]. JbossJTA provides distributed transactions through multiple resources, standards-based transaction support to J2EE applications, and protects against data corruption by ensuring complete, and accurate transactions for Java-based applications [10].

In this dissertation, we used the Arjuna library to implement the sample of transaction applications, because it provides transaction support for nested transactions and concurrent control notations.

11.2 Reuse and Modularization of Advanced Transaction Issues

Sarangdevot and Sharma investigate in the modular design of a real life distributed transaction applications in the domain of banking [14] and insurance [79][80] by using the AOP design methodology in an Eclipse-AJDT environment. The authors illustrate how separation of concerns is one of the key principles of good transaction software system design and implementation. Successful implementation of transaction applications conclude that the AOP methodology in the Eclipse-AJDT environment offers powerful support for modular design of transaction software systems. In the end, it is concluded that the AspectJ implementation is improved several software quality factors such as modularity, readability, understandability, maintainability, correctness, extendibility, reusability, traceability, flexibility, predictability, adaptability and ease of evolution. Reduction in development time and costing were also perceived. Thus, overall improvements in the quality and performance of the software system are realized [79].

Fabry pursues to provide a successful modularization of advanced transaction models (ATMS, sometimes also called extended transaction models) [81]. He proposes a new aspect language, *KALA*, which is a domain-specific aspect language for using advanced transaction management in a distributed system. It allows the modularization of the different concerns contained within such an ATMS and allows programmers to express their needs at a higher-level abstraction than what is accomplished with general-purpose aspect languages. As a result, this domain-specific aspect language significantly raises the level of abstraction and makes the demarcation code much more concise [81][115]. In addition, Farby provides two technical contributions: a general Transaction Processing Monitor based on the ACTA formal model that supports a wide variety of ATMS; and offers an implementation of an aspect weaver for the KALA language, which generates code using this interface.

TransJ provides many abstraction aspects (base-aspects), which can be extended to build a high-level of abstractions (reusable and application-level aspects) that can encapsulate more complex types of transaction concerns. *TransJ* also provides a central transaction process trackers in order to communicate with transaction logic (demarcation code) to achieve the better separation of concerns, and a high-level obliviousness. As opposed to the KALA, the *TransJ* is more about modeling transaction concerns. It defines a transaction joinpoint model, i.e., UMJDT, and reusable transaction abstractions in AspectJ language.

In [48][82], Cunha and Ekwa investigate approaches for implementing reusable aspects for high-level concurrency techniques in AspectJ. The authors present two alternate implementations can be used: one based on traditional pointcut interfaces and another based on annotations. Cunha illustrates these implementations showed how abstract pointcuts interfaces and annotations can be used to implement a well-defined high-level concurrency pattern and mechanisms, namely one-way calls, futures, waiting guards, readers/writers, barriers and active object; and use of AspectJ enables the development of reusable implementations of the above-mentioned approaches, in the form of higher modularity and unpluggability. In [83], the authors also compared the performance overhead, reusability and the unpluggability between conventional OOP implementations and AOP implementations. Eventually, they conclude that the AspectJ implementation is more reusable and pluggable, but incurs a noticeable performance overhead. However, AspectJ has a limitation in acquiring context and joinpoint information on concrete aspects: when a super-aspect defines an abstract pointcut, the sub-aspects cannot change the pointcut's signature. The ideas are similar in the sense that the *TransJ* weaves aspects into transaction applications using the AspectJ. We believe that in using *TransJ* the same level of concurrency patterns can be redefined in a more modular and oblivious fashion. It also allows developers to provide concrete pointcuts for each pointcuts to have their application advised-hindering obliviousness. It allows many reusable aspects, which can be extended to build more useful application-level aspects of concurrency control approach or other transaction-related concepts. In short, *TransJ* the only work required by developers to extends the functionality provided by reusable aspects library are to bind their application classes to the appropriate aspects. This requires no knowledge of the inner working of *TransJ*.

In [32] and [94], Douence, et al., present *Concurrent Event-based AOP* (CEAOP), which defines the approach of writing concurrent aspects. The authors provide an intuitive and simple model for sequential AOP, whose notion of aspects, defined in a model for concurrent aspects which extends the sequential event-based AOP approach. This model explicitly addresses the three AO-specific coordination issues: aspects modularize functionalities that typically modify base executions at a large number of execution points; advices can be divided into pieces that can be matched differently with the base execution; and multiple advices may apply at the same of execution point. Then it shows how to compose concurrent aspects using a set of general composition operators and sketches its Java prototypical implementation. The way paper tries to compose concurrent aspects shares some similarity with *TransJ*, however, its scope is more towards covering concurrency pattern, which is not the main research direction in our work (i.e., transaction).

Rashid, et al., [49] represents an effort to support persistence by implementing a reusable, modular and oblivious AO framework in AspectJ. The authors explored three issues in the context of AOP and data persistence: the possibility of using AOP techniques in aspectizing persistence, the reusability of persistence aspects, and whether persistence aspects could be developed independently of an application. However, application developers could only be partially oblivious to these aspects because of continuing to take into account the architectural decision during the design phase.

The scope of the persistence framework is very limited as compared to the *TransJ*. *TransJ* relies on well-defined both reusable and base aspects that can be used in a non-database persistent context, i.e., transaction context. The developers can reuse joinpoints and pointcuts for each of the transaction-related pointcuts in the reusable and base aspects to have their applications advised-interfering with obliviousness. *TransJ* applies these ideas into the domain of transactions, wherein the reusable aspects could be used to configure of application-level aspects of different transaction systems to modularize several types of transaction-related concerns.

11.3 Works on Joinpoint Model with Reference to *TransJ*

The Ph.D thesis [85] of Sadat-Mohtasham aims to provide a design, and implement *Transactional Pointcuts* as a realization of the new model in the AspectJ language. The contributions made by the thesis can be broken down into three different parts. First, the thesis proposes a new joinpoint model, based on the pointcut-advice model. The pointcut-advice model is a dynamic joinpoint model in which joinpoints are points in program execution. This model takes joinpoint inter-relationships into account and allows the designation of more complex computations as joinpoints. In other words, it makes designation and advice of interrelated joinpoints possible. Secondly, on top of this model, the thesis designs a new construct, *transcut*, which selects sets of interrelated joinpoints and reify them into higher-level joinpoints that can be advised. Third, the thesis shows a design a *transcut* matching algorithm based on single-entry single-exit regions of control dependences in a control flow, i.e., the *Program Dependence Graph* (PDG). This representation is the backbone of the *transcut* matching algorithm.

Alongside the above research contributions, the thesis also provides a technical contribution. Based on the *transcut* it presents an AspectJ language extension that realizes transactional pointcuts. The author has extended abc’s existing joinpoint matching infrastructure for *transcut* matching by implementing the appropriate subclasses (for the new type of shadow, new pointcuts, etc.,) and by advising the right joinpoints to adapt the behavior of some of the existing components in the context of *transcut* matching. If a *transcut* matches a shadow, an advice application object is created to be applied to the shadow in the weaving phase. All three major types of advice (i.e., around, after, and before) are supported for *transcuts*. There are some differences between transactional pointcut model and our work. Transactional pointcut relies on static analysis only and, therefore, is inherently imprecise. Our model uses an interval joinpoints (execution-time joinpoint model) to determine dynamically when an advice should stratify. Also, the *TransJ* designation and advice model complies with the existing dynamic pointcut-advice model in AspectJ, which made it possible for integration and interaction with an AOPL, such as AspectJ.

On the other hand, the author discusses dynamic meta-model annotations to add well-separated concerns. He shares some design similarities for *TransJ* that is joinpoints in *transcut* is identified as part of a bigger context and in relation to other joinpoints. *TransJ* design principles include a similar concept for implementing transaction patterns using AspectJ, but the *TransJ* handles transactions in high-level transaction abstractions rather than low-level abstractions. It allows to encapsulate the transaction concerns from core application functionality with writing reusable and application-level transaction aspects as explained in *TransJ*, Chapters 4, 5 and 6. In addition, it already provides a set of reusable aspects and have the ability to compose the application-level aspects.

Rajan et al. [86], propose a new language, *Ptolemy*, which adds quantified, typed events to implicit invocation languages, producing a language that has many of the benefits of both implicit invocation and AOPLs. A drawback of implicit invocation languages is their incapability to refer to a large set of events succinctly. They also lack the expressive power of AOP advice. Limitations of AOPLs include potentially fragile dependence on syntactic structure that may hurt reusability and maintainability, and limits on the available set of implicit events and the reflective context information available. The authors implement the Quantified, typed events in their language Ptolemy to solve all these problems. Ptolemy lets explicit declaration of event types that have a name and a set of variables used to expose context. Arbitrary sequences of expressions can be annotated as having a specific event type. Objects can register to be able to advise the event types they are interested in. When an event of a specific type is fired, all the relevant advice is executed. The designers of Ptolemy also recognize the need for arbitrary pieces of code to be treated as typed joinpoints. In Ptolemy, the events have to be explicitly announced in the target code while our objective is obliviousness. That is, *TransJ* allows event type declarations in an implicit manner. This implies that programmers are completely oblivious of the inner workings of the *TransJ*. Therefore, it already provides a set of reusable and application-level aspects and have the ability to compose using base aspects.

Harbulot, et al. [87], propose an AspectJ language extension, called *LoopsAJ*. In LoopsAJ, the authors add a primitive pointcut loop that can match loops in the program and can expose values like the minimal and maximal value of the loop-iteration counter and its stride. The authors define a loop pointcut to designate and advice loops, which is in essence, is an arbitrary computation as joinpoints that are referenced to through their key constituent joinpoints. Although loops are an instance of such computations, the loop pointcut cannot select loops based on what they do, instead, all found loops are exposed. LoopsAJ's performs loop matching on the bytecode level, and it can recognize well-structured loops only.

Bodden introduces *Closure Joinpoints* (CJs) [88], as a design for explicit block joinpoints that yields a syntax and semantics close to the JPL. In writing CJs, he proposes to implement an “extract joinpoint" refactoring, similar to the traditional “extract method" refactoring. But he did not show an integration of CJs into the AspectJ.

Akai et al., propose region pointcuts for AspectJ [37][89]. Region pointcuts consist a “region match pattern" over regular AspectJ pointcuts. This allows developers to select regions, i.e., intervals of time that start when matching one regular AspectJ pointcut and end when matching another. Region pointcuts are quite powerful in that they give aspect programmers very fine-grained control about which statements exactly constitute a joinpoint. On the other hand, region joinpoints may increase the problem of fragile pointcuts: because region pointcuts are very explicit about syntactic constructs of the base program, and even the order in which they occur, they may increase coupling to the particular base code at hand. Explicit joinpoints circumvent this problem by assuming that the core-code programmer is aspect-aware and includes relevant joinpoints in the core code. Although region pointcuts pick out implicit logical intervals of the core code, these intervals are nevertheless subject to the very control-flow and data-flow constraints. The idea is similar in the sense that *TransJ* defines joinpoints, which corresponds to a logical interval of time in a flow of execution. It has a beginning and an end, and advice can be woven into the flow of execution before, after, or around it. In other words, these joinpoints define the region of code, where advice may be woven.

Nishizawa, et al., propose a remote pointcut and remote inter-type declaration as an extension to AspectJ language for distributed software [74]. The language construct, *called remote pointcut,* enables developers to write simple aspects to modularize crosscutting concerns related to distributions, scattered on multiple hosts. Similarly, Pawlak presents a framework to build AO distributed applications in Java [90]. He talks about dynamic wrappers (also called *generic advice*) and meta-model annotations to add well-separated concerns. The author also provides a way to define distributed pointcuts. With similar concept, Kaewkasi [91] proposed distributed advice code execution. The interesting idea is the distributed advice execution using shared execution units. Along the similar lines, Mondejar [92] introduces a complete aspect remoting service with 1-to-1 and 1-to-*(many) abstractions, outlines a distributed joinpoint model to intercept remote services. The notion of remote service abstractions such as 1-to-1 and 1-to-* abstractions and later its implementation as any pointcut, many pointcuts and multi-pointcuts share some design principles with our work. These papers share some design similarities and future extension points for *TransJ*.

11.4 Other Work on Interesting Crosscutting Concerns and Design Principles with Reference to *TransJ*

Kiczales, et al. [3], introduced the idea of weaving logic for crosscutting concerns into core applications was introduced over 15 years ago, and their work stems from even earlier research with inheritance, aggregation, and mix-ins [2]. Like all great ideas, the heart of the weaving solution is relatively straightforward – modularize concerns into first-class constructs, find the right place(s) to introduce appropriate logic from those constructs, and the either insert code that executes the new logic unconditional (because it can be determined to always be needed) or insert code that makes a final decision about executing the new code at runtime. Raza, et al., present the design and implementation of a new AOPL framework, called *CommJ*, which is an extension to AspectJ for enabling programmers to encapsulate communication-related crosscutting concerns in modular, cohesive and loosely coupled aspects [31]. *CommJ* allows developers to weave crosscutting concerns into *inter-process communications* (IPC) in a modular and reusable way, while keeping the core functionality oblivious to those concerns. They also discuss an initial study on hoped-for benefits of *CommJ* in comparison with AspectJ [30]. It does so by evaluating certain desirable characteristics defining a quality model that can be measured by computable metrics. Based on initial theoretic notions, they hypothesize that developers should see reuse and maintenance improvements relative to six desired qualities defined by the quality model.

The results from this preliminary investigation provides sufficient evidence to conclude that *CommJ* is capable of encapsulating a wide range of communication-related crosscutting concerns and that it can provide better maintainability and reusability. This is in many respects, we found some conceptual similarity with this design approach to our work, but we have a different goal in that it addresses how to weave transaction-related crosscutting concerns into high-level runtime abstractions, i.e., distributed transactions. *TransJ* framework provides low-level distributed aspects that perform the expected weaving and tracking of context information. We believe this to be feasible because it is similar to the technique used by CommJ to add communication-related aspects to AspectJ. It also clearly defines transaction primitives for the DTPS that defines interesting joinpoints relative to transaction execution and related contexts for the woven logic of crosscutting concerns. We believe our work paves the way for the weaving of crossing cutting concerns into high-level program abstractions that span multiple threads of execution and may be interleaved with concurrent execution of similar the abstraction. It also can define more reusable aspects, which not only can be extended, but can also be combined to build more complex types of transaction concerns.

The main contribution of the Soares, et al., in their paper [36], is to provide architectural guidelines and implementation of several persistence and distribution concerns in the application using AspectJ. The authors demonstrate that coding crosscutting concerns using AspectJ are a better option than to write in plain Java language. This paper shares some architectural guidelines with *TransJ* architecture. In addition, Soares, et al. [103], define guidelines to reorganize OO software in order to modularize concurrency control using AOP. Those guidelines are supported by a concurrency control implementation that guarantees system correctness without redundant logic of concurrency control, both increasing software efficiency and guaranteeing safety. The author provides aspect framework that offers simple aspects that can be reused to implement the logic of concurrency control in other applications, thus it makes the concurrency control easy to develop and reduces the complexity of other related components of the software system, such as business and data management modules, by decoupling the logic of concurrency control code from core applications.

Netinant describes an AO framework where both functional components and system properties are designed relatively separate from each other [95]. This framework allows developers to build software systems that are manageable, stable and adaptable. Most of the work in this paper concentrates on the decomposition of concurrent OO systems with the objective to accomplish a better separation of crosscutting concerns in both design and implementation. It highlights the general design principles of separation of concerns, some of which can be employed in *TransJ* to improve its existing design.

Perhaps one of the most relevant previous work to our research is trace-based aspect techniques in which joinpoints are runtime execution events and pointcuts consist of patterns of events [93][94]. Among these mechanisms, Tracematch [93] seems to be the most developed and has been added as an extension to the *abc* compiler. It includes three parts: an event definition part that defines the runtime events of interest using pointcuts; a regular pattern of the defined events; and advice to be executed when the pattern is matched at runtime. Runtime events are tracked and once a specified pattern of events is occurred, an advice can be executed. Similarly, *TransJ* uses joinpoint trackers, which are monitors that perform pattern matching on transaction events, to track events and organize them into high-level transaction contexts.

The methodologies cited above do not seem to be tailored to satisfy the software traceability for DTAs. Therefore, what makes our work different compared to other works in this area is that the fact that we explicitly deal with transaction-related crosscutting concerns and providing a reusable aspect library.

11.5 Measurement Metrics with Reference to EQMTA

Within software engineering, there are software quality models made for quantifying various qualities of the attributes such as Boehm’s Model, and McCall's Model [96]. McCall proposed quality factors in the early 1970s. They are as valid today as they were at that time. This model was developed to measure the relationships between external factors and product quality criteria. It started with a volume of 55 quality characteristics that have an important effect on software quality, and called them *factors*. The quality attributes were classified into three major varieties, eleven factors that describe the external view (user view) of the software system, twenty-three internal attributes which describe the internal view (developer view) of the software system, and metrics which defined and used to offer a scale and methodology for measurement. In our experiment’s perspective, we select reusability and performance as the most important quality factors to consider initially because of the potential for cost savings they both represent. Further work could focus on some of the other nine factors. Boehm added new quality factors to McCall’s model with concentrating on the maintainability of software systems. The goal of this model is to address the contemporary deficiencies of models that quantitatively and automatically calculate the quality of software system. Therefore, the Boehm model characterizes the features of the software system hierarchically in order to get contribute in the total quality [101].

The previous research work contains several sets of traditional metrics and others for OO software systems. Most existing traditional metrics cannot be applied straightforwardly to AO software system [97], since AOP introduces new abstractions to software engineering. Up to now, most empirical studies in the AO context on subjective criteria and qualitative investigation [61][65][68][98][99]. For example, Sant’Anna’s quality model [54][55] presents a suite of metrics and a quality model, which supports different kinds of implementation environments, to evaluate the assessment of AO software in terms of reusability and maintainability. The author provides the Quality model [54] using Basili’s Goal-Question-Metric (GQM) Methodology [61]. Basili offers a three-step framework: (1) list the key Goals of the empirical study; (2) derive from each goal the Questions that must be answered to define if the goals have been met; (3) decide what must be measured in order to be able to answer the questions effectively. The proposed metrics satisfy key requirements in order to realize successful measurements in the AO context. These metrics are more generalized to measure different concerns of design and code as compared to Lopes’ work [59]. Moreover, its metrics are early prediction mechanisms for different stringent principles within the design of AO software system, like coupling and cohesion. Therefore, Sant'Anna's quality model is strong enough to be applied to different types of implementations. Some other metrics [61] can be considered as complimentary to our chosen quality model, but they are not based on well-known software engineering quality models.

We use these models to define a new extended model for transactional application. It defines new quality internal attributes and metrics, such as extensibility, scalability, predictability, transaction volume, transaction velocity, etc., as discussed in Chapter 7.

Raza and Clyde define a reuse and maintenance quality model as an extension to Sant' Anna Quality Model [54], called *Extended-Quality Model* (EQM) [30]. The authors add internal attributes such as, the localization of design decisions, and code obliviousness. The EQM includes sixteen metrics for the six different internal attributes. Tools can compute ten of the metrics for code written by the subjects. The others have to be computed by hand.

Our quality model is based on these models and made a few enhancements to the EQM [30] and hope that doing so would further strengthen the model. The quality of transaction system is defined according to two major perspectives: product revision (ability to undergo change and adapt to new application) and product performance (its transaction efficiency). For instance, the interpretation of reusability and performance is dependent upon understandability and efficiency factors. As per our definitions of the qualities (see Chapter 7), code obliviousness, localization of design decisions, extensibility, efficiency, predictability, and scalability, are very important factors in the model. Further, Raza, Parnas, and Coady previously defined properties of modular code as being flexible, comprehensive, code obliviousness and independent development [30][61][65]. At that time, code predictability, scalability, extensibility were not the primary concern, but became an important element of software design in later years after emerging research in AOSD.

In our model (EQMTA), using a combination of qualitative and quantitative approaches can improve an assessment by ensuring that the restrictions of one type of data are balanced by the strengths of another. In other words, we rely on the positivistic approach, using quantitative methods, such as quality metrics, but others, such as qualities, quality factors, and quality attributes, are of qualitative is that it usually depend on inductive reasoning.

CHAPTER 12

SUMMARY AND FUTURE WORK

12.1 Summary

This dissertation took the necessary steps to introduce the notation of transaction-aware aspects to incorporate transaction-related crosscutting concerns into an AspectJ framework, namely *TransJ*. *TransJ* is an independent abstraction framework that uses aspects as main abstractions and proposes a model for distributed transaction aspects and transaction joinpoints for weaving crosscutting concerns into transaction abstractions. In other words, it allows developers to encapsulate transaction-related crosscutting concerns in reusable modules.

Since the transaction-related concepts, such as transaction itself, transaction operation, concurrency control, etc., cannot encapsulate the crosscutting concerns, the first step was to propose a new conceptual model, i.e., UMJDT, to define interesting joinpoints relative to transaction execution and context data for woven advice. Each joinpoint referred to a specific context, e.g., transaction context, operation context, or lock context. The context is basically a region (interval) of computation with well-defined information, make it a perfect choice for encapsulating the transaction-related crosscutting concerns.

With the introduction of UMJDT model, it then describes the design and implementation of some of *TransJ* components, such as the base aspects, each one providing a well-defined reusable functionality. It also provides an overview of a toolkit, i.e., the reusable library that consists of reusable transaction aspects and doubles as a proof of concepts, since these aspects can be directly applied to a wide range of existing transaction applications. It then shows how these reusable aspects can be configured and composed in different ways to encapsulate new concerns.

We believe that *TransJ* is capable of encapsulating a wide range of transaction-related crosscutting concerns in aspects. We hope to gather more empirical evidence of the *TransJ*'s value by increasing the number of aspects in the reusable aspects and by continuing to expand the number and types of applications that use *TransJ*.

We also conducted a research experiment to compare AspectJ with *TransJ* for various software design attributes related to reuse and performance through an extended-quality model for transactional application, namely EQMTA. Initial findings from this experiment revealed that crosscutting concerns programmed in *TransJ* delivered more modular, reusable programs without sacrificing the performance. However, our future research will include more formal software-engineering productivity experiments to verify the performance belief.

A primary conclusion of this dissertation is that the use the *TransJ* joinpoints to model transaction-related crosscutting concerns facilities the creation of reusable and application-level aspect library. The base aspect increases code reuse and reduce pointcuts complexity. Achieving a semantic separation of concerns while also achieving highly reusable aspect components. The abstract base and reusable aspects must be wisely designed and implemented to be as minimal as possible or whole application reusability may actually decrease. In *TransJ*, replacing the actual method with a set of specifications that make the access to the base application code unnecessary and allow aspect's advice to be woven into base aspect at runtime that help to maintain the runtime performance. Finally, the greatest application reusability and extensibility are achieved when a combination of transaction joinpoints and oblivious aspects are applied and when having a stable fashion of pointcuts. Therefore, the use *TransJ* result in better code quality.

12.2 Future Work

As with any research study, this work conceptually opens avenues for future work. We envision a number of extensions or spins off to *TransJ*.

- Package *TransJ* and release as an open source extension to AspectJ.
- *TransJ* has the potential to be very useful for testing various kinds of transaction models related errors in transaction abstractions. We plan to explore this potential and additional experiments focus on analyzing the exception handling strategies in these models and providing them into reusable aspects.
- The performance metrics show a slight improvement in the performance of transaction applications are implemented on *TransJ*. Therefore, we plan to explore this improvement and make additional experiments in different types of transaction systems in different environments.
- We plan to conduct additional studies, refine the *TransJ* Infrastructure and extend the library of reusable aspects.

- A future work that may bring a new dimension to the *TransJ* is the introduction of dependencies among the contexts. Currently, except the nested transaction model, there is no relationship between different contexts. Therefore, we need to handle various kinds of dependencies among the contexts to improve the coupling and cohesion between reusable aspects.
- *TransJ* framework could also handle models such as recovery. To achieve this as an ambitious goal, the framework must be well analyzed to determine what additional aspects have to be incorporated into *TransJ*.
- *TransJ* can be extended for distributed remote pointcuts that would simplify the implementation of even more complex crosscutting concerns, such as recovery, or multithreaded in a distributed system.
- Finally, *TransJ* has potential to support different kinds of transaction managers, such as, bitronix, atomikos, or spring transaction manager. We plan to explore this potential and additional experiments focus on implementing strategy pattern for different types of trackers.

REFERENCES

[1] F. P. Jr. Brooks, “No silver bullet, essence and accidents of software engineering”, *IEEE Computer*, vol. 20, no. 4, pp.10 -19, 1987.

[2] G. Booch and R. M. Maksimchu, “Object-Oriented Analysis and Design with Applications.” *Addison-Wesley Professional*, third edition, Boston, Published April 1st 2007.

[3] G. Kiczales et al., “Aspect-Oriented Programming.” *Proceedings of ECOOP '97*, Springer Verlag, pages 220--242, 1997.

[4] J. Gradecki and N. Lesiecki, “Mastering AspectJ.”, *Wiley Publishing Inc*, 456 pages, March 2003. ISBN: 978-0-471-43104-6.

[5] L. Rosenhainer, “Identifying Crosscutting Concerns in Requirements Specifications.” *In Proceedings of OOPSLA Early Aspects*. October 2004, Vancouver, Canada. 2004. http://trese.cs.utwente.nl/Docs/workshops/oopsla-early-aspects-2004/

[6] C. Clifton and G. T. Leavens, “Obliviousness, Modular Reasoning, and the Behavior Subtyping Analogy”, *In Proceedings of the Workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT)*, Workshop at AOSD, December 2003, pp.1-6.

[7] 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.

[8] Eclipse, 2015, Sep 30, AspectJ, [Online]. Available: http://www.eclipse.org/aspectJ/

[9] Aspectwerkz2, 2015, Aug 10, Plain Java AOP, [Online]. Available: http://aspectwerkz.codehaus.org/

[10] Jboss, 2015, Jul 30, JBoss AOP, [Online]. Available: http://www.jboss.org/jbossaop

[11] Spring, 2015, Sep 30, Spring AOP, [Online]. Available: http://www.springframework.org

[12] A. Rausch et al., “Aspect-Oriented Framework Modeling.” *4th AOM Workshop at UML’03*, San Francisco, CA, Oct. 2003.

[13] A. Kaur and K. Johari, “Identification of Crosscutting Concerns: A Survey.” *International Journal of Engineering Science and Technology* 1, vol. 3, 2009, pp. 166-172.

[14] S. Sarangdevot and A. Sharma. “Investigating the application of AOP methodology in development of Banking Application Software Using Eclipse-AJDT environment.” *Journal of Management Sciences*, AIMS, Udaipur, Rajasthan, 2009. pp. 124-141.

[15] G. Kohad et al., “Concept and techniques of transaction processing of Distributed Database management system,” *International Journal of Computer Architecture and Mobility*, Volume 1-Issue 8, June 2013. ISSN 2319-9229.

[16] J. Gray, “The Transaction Concept: Virtues and limitations”, *In Proceedings of the 7th International Conference on VLDB Systems*, ACM, New York, 1981, pp. 144-154.

[17] J. Gray and A. Reuter, "Transaction Processing: Concepts and Techniques." *The Morgan Kaufmann Series in Data Management Systems*. Morgan Kaufmann, San Mateo, CA, 1993. ISBN-13: 978-1558601901, ISBN-10: 1558601902.

[18] A. AlSobeh and S. Clyde, "Unified Conceptual Model for Joinpoints in Distributed Transactions." *ICSE'14. The Ninth International Conference on Software Engineering Advances.* Nice, France. October, 2014. ISBN: 978-1-61208-367-4.

[19] Java Community Process, 2005, Java Specification Requests [Online]. Available: http://encyclopedia.thefreedictionary.com/Java+Community+Process

[20] Sun Microsystems Inc. 1999. Java Transaction API (JTA) [Online]. 901 San Antonio Road, Palo Alto, CA 94303. Available: https://www.progress.com/jdbc/resources/tutorials/understanding-jta.

[21] JBoss. 2006. *JBoss Transactions API (JBossJTA)* 4.2.3 [Online]. Home of Professional Open Source Copyright 2006, JBoss Inc. Available: http://docs.jboss.org/jbosstm/docs/4.2.3/manuals/pdf/jta/ProgrammersGuide.pdf.

[22] Narayana, 25 Sep 2015, Narayana Transaction Anaylser (NTA) Tool [Online], Available: http://narayana.jboss.org/

[23] Atomikos, 2015, ExtremeTransactions Atomikos Transactions [Online]. Available: http://www.atomikos.com

[24] Bitronix , 2015, The Bitronix JTA Transaction Manager [Online], Available: http://www.bitronix.be/

[25] Office of Research and Graduate Studies at Utah State University, (2015, Jun 10) Institutional Review Board [Online]. Available: http://rgs.usu.edu/irb/.

[26] I. Sommerville, "Software Engineering", 8th ed., *Addison-Wesley*, Boston, Pearson Education Limited. 2009. ISBN-13: 978-0137035151, ISBN-10: 0137035152.

[27] R. Laddad, "AspectJ in Action." *Manning Publication Co.*, Greenwich, CT, pages 512. Jul. 2003. ISBN 9781930110939

[28] J. Kienzle, R. Guerraoui. "AOP: Does it Make Sense? The Case of Concurrency and Failures." Proc. ECOOP'02, pp.37-61, June 10-14, 2002.s

[29] G. Kamble, "Aop - Introduced Crosscutting Concerns." *Proceedings of 2009 International Symposium on Computing, Communication, and Control. Singapore,* 9-11 October, 2009, pp. 140-144. ISBN 978-9-8108-3815-7.

[30] A. Raza and S. Clyde, "Communication Aspects with CommJ: Initial Experiment Show Promising Improvements in Reusability and Maintainability," ICSEA 2014. Nice, France, Oct. 2014.

[31] A. Raza and S. Clyde, "Weaving Crosscutting Concerns into Inter-process Communications (IPC) in AspectJ," *ICSEA 2013*. Nov. 2013, Venice, Italy, pp. 234-240. ISBN: 978-1-61208-304-9.

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

[33] G. Kiczales et al., "An Overview of AspectJ". *15th ECOOP 01*, June 2001, pp. 327 – 357.

[34] Aspectc, 16 Sep. 2015, AspectC++ [Online], Available: http://www.aspectc.org/

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

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

[37] S. Akai and S. Chiba, "Extending AspectJ for separating regions". *In GPCE,* pages 45-54. ACM, 2009.

[38] G. Alkhatib and R. S. Labban, "Transaction Management in Distributed Database Systems: the Case of Oracle's Two-Phase Commit," *The Journal of Information Systems Education*, vol.13:2, 1995, pp. 95-103.

[39] T. Härder and K. Rothermel, "Concurrency Control Issues in Nested Transactions," *Journal of VLDB 2 (1),* Jan. 1993, pp. 39-74.

[40] Redhat, 2007, Jboss Transaction Manager [Online], Available: http://docs.jboss.org/jbosstm/docs/4.2.3/manuals/pdf/jta/ProgrammersGuide.pdf,

[41] C. Mohan et al., "Transaction Management in the R*· Distributed Database Management System," *ACM Trans. on Database Systems*, vol. 11, no. 4, pp. 378-396, December. 1986.

[42] G. Colouris et al., "Distributed systems, concepts and design," *Addison-Wesley,* Fourth edition, 2005. ISBN-10: 9780321263544, 2005.

[43] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys Vol*. 13 No 2, June, 1981, pp. 185-221.

[44] B. Dibyendu, 23 May 2001, J2EE Transaction Frameworks: Distributed Transaction Primer [Online], Available: http://www.onjava.com/pub/a/onjava/2001/05/23/j2ee.html

[45] S. Mahapatra, 14 Jul 2000, Transaction Management under J2EE 1.2 [Online], Available: http://www.javaworld.com/article/2076126/java-se/transaction-management-under-j2ee-1-2.html.

[46] Spring, 2001, Transaction Management [Online], Available: http://docs.spring.io/spring-framework/docs/4.0.x/spring-framework-reference/html/transaction.html

[47] Narayana, 2015 , Jboss Narayana Documentation Library [Online], Available: http://narayana.jboss.org/documentation/5_0_2_Final

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

[49] A. Rashid and R. Chitchyan, "Persistence as an aspect." *In: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development- Development (AOSD*), Boston, MA, USA, March 2003, pp. 120–129. ACM, New York, NY, USA. 2003.

[50] J. Kienzle et al., "AspectOptima: A Case Study on Aspect Dependencies and Interactions." *Trans Aspect-Oriented Softw Dev 5:187–234, Springer-Verlag,* Berlin, Heidelberg, 2009. [doi>10.1007/978-3-642-02059-9_6].

[51] K. Sirbi and P. J. Kulkarni, "Enhancing Modularity in Aspect-Oriented Software Systems-An Assessment Study," *(IJCSE) International Journal on Computer Science and Engineering,* Vol. 02, No. 06, 2010, 2014.

[52] A. Sharma, S. Sarangdevot, “Eclipse-AJDT Environment: A Diamond from Open Source Technology,” *In Proc. International Conference on Next Generation Communication and Computing Systems (ICNGC2S-10)*, December 25-26, 2010, Chandigarh, India, pp. 120-125.

[53] R. Filman and D. Friedman, “Aspect-oriented programming is quantification and obliviousness.” *In OOPSLA Workshop on Advanced Separation of Concerns*, Minneapolis, USA, October 2000.

[54] C. Sant’Anna et al. “On the Modularity Assessment of Aspect-Oriented Multiagent Architectures: a Quantitative Study.” *International Journal of Agent-Oriented Software Engineering*, v. 2, pp. 34-61, 2008.

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

[56] Wikipedia. 2015, March. 28. Block Diagram. [Online]. Available: http://en. wikipedia.org/wiki/Block_diagram.

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

[58] A. Garcia, “On the Modular Representation of Architectural Aspects.” *In: Gruhn, V., Oquendo, F. (eds.) EWSA 2006*. LNCS, vol. 4344, pp. 82–97. Springer, Heidelberg. 2006.

[59] C. Lopes et al., “Sushil Krishna Bajracharya, Assessing aspect modularizations using design structure matrix and net option value,” *Transactions on Aspect-Oriented Software Development I, Springer-Verlag*, Berlin, Heidelberg, 2006.

[60] A. Rashid, M. Aksit (Eds.), Transactions on Aspect-Oriented Software Development (TAOSD), *In: LNCS* 3880, April 2006, pp. 1–35.

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

[62] 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.

[63] A., Garcia et al., “Engineering Multi-Agent Systems with Aspects and Patterns”. *J. of the Brazilian Computer Society*, July 2002, 1 (8), pp 57-72.

[64] A. Garcia et al., (2003). *Agents and Objects: An Empirical Study on Software Engineering*. Technical Report 06-03, Computer Science Department, PUC-Rio, February 2003. Available FTP:ftp://ftp.inf.puc-rio.br/pub/docs/techreports/ (file 03_06_garcia.pdf)

[65] Y. Coady et al., “Can AOP support Extensibility in Client-Server Architecture?” *In European Conference on Object-Oriented Programming (ECOOP)*, Aspect-Oriented Programming Workshop, June 2001.

[66] Collaborative Institutional Training (CIIT), 2014, Social & Behavioral Research Modules [Online], Available: https://www.citiprogram.org. Last updated on 11/11/2014.

[67] C. Sant’Anna et al., “On the Modularity of Software Architectures: A Concern-Driven Measurement Framework.” *In Proc. ECSA*, 2008. 24-26, 2007, Madrid, Spain.

[68] R. Burrows et al., “Coupling metrics for aspect-oriented programming: A systematic review of maintainability studies,” *In Evaluation of Novel Approaches to Software Engineering*, volume 69 of Communications in Computer and Information Science, pp. 277-290. 2010.

[69] 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.

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

[71] G. Jay, et al., "Cyclomatic complexity and lines of code: empirical evidence of a stable linear relationship," *Journal of Software Engineering and Applications*, vol. 3, no. 2, pp. 137-143, 2009.

[72] SourceForge, 2015, Jul 10, Eclipse Metrics Project 1.3.6 [Online]. Available: http://metrics.sourceforge.net

[73] E. Figueiredo et al., “AJATO: An AspectJ assessment tool,” *In Proc. 20th ECOOP*, Nantes, France, 2006. Nantes, France. July 3-7, 2006.

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

[75] M. Marin et al.”SoQueT: Query-based documentation of crosscutting concerns”. *In Proc. 29th Intl. Conf. on Software Engineering (ICSE)*. IEEE Computer Society, 2007.

[76] L., Rosenhainer, “Identifying Crosscutting Concerns in Requirements Specifications.” *In Proceedings of OOPSLA Early Aspects* 2004. October 2004, Vancouver, Canada.

[77] J. Kienzle and S. G´elineau, “Ao challenge - implementing the acid properties for transactional objects,” *In AOSD ’06: Proceedings of the 5th international conference on Aspect-oriented software development*, pp. 202–213, New York, NY, USA, 2006. ACM Press.

[78] K. Panos et al., “Acta: A framework for specifying and reasoning about transaction structure and behavior,” *In SIGMOD Conference*, pages 194–203, 1990.

[79] A. Sharma and S. Sarangdevot, “Application of FOP and AOP Methodologies in Concert for Developing Insurance Software Using Eclipse-Based Open Source Environment,” *Communications in Computer and Information Science, Springer-Verlag Belin,* pp. 592-606, 2011.

[80] A. Sharma, S. Sarangdevot, “Investigating the Application of AOP Methodology in Development of Insurance Application Software Using Eclipse-AJDT Environment,” *In Proc. International Conference on Computer Engineering and Technology (ICCET’10)*, Nov. 13-14, 2010, Jodhpur, India, pp. D-17 – D-25. 2010.

[81] J. Fabry et al., “KALA: Kernel Aspect Language for Advanced Transaction,” *Science of Computer Programming*, pp. 165-180, 2008.

[82] J. Ekwa and D. Ekoko, “Evaluating the expressivity of aspectj in implementing a reusable framework for the acid properties of transactional objects.” Master’s thesis, School of Computer Science, McGill University, Montreal, Canada, 2006.

[83] G. Bolukbasi, “Aspectual Decomposition of Transaction,” Master’s thesis, School of Computer Science, McGill University, Montreal, Canada, 2007.

[84] K. Hoffman and P. Eugster, "Trading Obliviousness for Modularity with Cooperative Aspect-Oriented Programming," *ACM Transaction Software Engineering and Methodology*, pp. 120–129, 2013.

[85] H. Sadat-Mohtasham and H.S. Hoover, H.S, "Transactional pointcuts: designation reification and advice of interrelated join points," *In Proceedings of the 8th International Conference on Generative Programming and Component Engineering*, pp. 35–44. ACM, October 2009.

[86] H. Rajan and G. Leavens, "Ptolemy: A language with quantified, typed events," *ECOOP 2008 – Object-Oriented Programming*, pp. 155–179, 2008.

[87] B. Harbulot et al., "A join point for loops in aspectj," *In AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 63–74, New York, NY, USA, 2006. ACM.

[88] E. Bodden, "Closure joinpoints: block joinpoints without surprises," in *Proceedings of the 10th ACM International Conference on AspectOriented Software Development (AOSD 2011)*. Porto de Galinhas,Brazil: ACM, Mar. 2011, pp. 117–128.

[89] S. Akai et al., "Region pointcut for AspectJ," *In ACP4IS '09: Proceedings of the 8th workshop on Aspects, components, and patterns for infrastructure software*, pages 43-48. ACM, 2009.

[90] R. Pawlak, et al., "Jac: An Aspect-Based Dynamic Distributed Framework", Software Practise and Experience (SPE), volume 34(12), pp.1119–1148, 15 JUN 2004. DOI: 10.1002/spe.605.

[91] C. Kaewkasi, and J.R., Gurd, "A Distributed Dynamic Aspect Machine for Scientific Software Development", *Workshop VMIL* 07, Vancouver, BC, Canada. 2007.

[92] R. Mondejar et al., "Building a Distributed AOP Middleware for Large Scale Systems", *Workshop NAOMI 2008*, Brussels, Belgium. 2008.

[93] C. Allan et al., "Adding trace matching with free variables to aspectj," *SIGPLAN Not., 40(10)*. pp. 345–364, 2005.

[94] R. Douence, et al., "Trace-based aspects. Aspect-Oriented Software Development," *Addison-Wesley*, pp. 201–217, 2004.

[95] P. Netinant, 2004. *Composition of System Properties* [Online], Report Bankok Univeristy. 2004. Available: http://www.bu.ac.th/knowledgecenter/epaper/jan_june2004/paniti.pdf

[96] J. McCall et al., "Factors in Software Quality", *Nat'l Tech. Information Service*, no. Vol. 1, 2 and 3. 1977.

[97] J. Zhao, "Towards a Metrics Suite for Aspect-Oriented Software," *Technical-Report SE-136-25, Information Processing Society of Japan (IPSJ)*, pp. 112-119, March 2002.

[98] U. Kulesza et al., "Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study". *In Proc. ICSM*, pp. 223-233, 2006.

[99] M. Ceccato, and P., Tonella, "Measuring the Effects of Software Aspectization," *WARE*, pp.118-124. 2004.

[100] J. Zhao, "Measuring Coupling in Aspect-Oriented Systems", *Int.Soft. Metrics Symp*. 2004.

[101] B. Boehm, Characteristics of software quality, 2nd edition, *North-Holland Pub. Co*., 1978 - Computers - 169 pages. 1978.

[102] L. Madeyski and L. Sza la, “Impact of aspect oriented programming on software development efficiency and design quality: an empirical study”, *IET Software Journal*, vol. 1, no. 5, pp. 180–187, 2007. http://dx.doi.org/10.1049/iet-sen:20060071

[103] S. Soares and P. Borba, "Towards reusable and modular aspect-oriented concurrency control," I*n Sac '07: proceedings of the 2007 acm symposium on applied computing*, pp. 1293-1294, 2007. (Pubitemid 47568480)

[104] E. Gamma et al., 3rd edition, Design Patterns- Elements of Reusable Object Oriented Software. *Addison-Wesley,* 1995.

[105] S. Gudmundson and G. Kiczales. “Addressing practical software development issues in AspectJ with a pointcut interface.” *In Workshop on Advanced Separation of Concerns of ECOOP’01*, 2001.

[106] M. Tatsubori, "Separation of Distribution Concerns in Distributed Java Programming," *Addendum to the 2001 Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001),* Doctoral Symposium, pp.19-20, Tampa Bay, Florida, USA, October 14-22, 2001.

[107] C. Schwanninger et al., "Encapsulating Crosscutting Concerns in System Software," *Proceedings of the Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, March 22, 2004.

[108] R. Douence, et al., “A Formal Definition of Crosscuts”, *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pp.170-186, September 25-28, 2001.

[109] J. Varia and S. Mathew, January 2014, Overview of amazon web services [Online], Available: http://aws.amazon.com/whitepapers;.

[110] A. Popovici et al., “Dynamic Weaving for Aspect Oriented Programming.”*1st Intl. Conf. on Aspect-Oriented Software Development*, Enschede, Netherlands, Apr.2002.

[111] A. Popovici et al., “Just-In-Time Aspects: Efficient Dynamic Weaving for Java.” *2nd Intl. Conf. on Aspect-Oriented Software Development*, pp. 100–109, Boston, USA, March 2003.

[112] A. AlSobeh and S. Clyde, “TransJ: Abstract Independent-Framework for Weaving Crosscutting Concern into Distributed Transactions”. Unpublished paper. To be submitted to *IJARCSSE*, 2015.

[113] A. AlSobeh and S. Clyde, “Transaction Aspects with TransJ: Initial Experiment Show Promising Improvements in Reusability while Maintaining the Performance.” Unpublished paper. To be submitted OOPSLA, 2016.

[114] S. Malek, H.R. Krishnan and J. Srinivasan, “Enhancing Middleware Support for Architecture-based Development through Compositional Weaving of Styles.” *J. Syst. Softw.* **83**(12), pp 2513–2527. 2010.

[115] Theo D'Hondt. "KALA", *Proceedings of the 2006 ACM symposium on Applied computing - SAC 06,* SAC 06, 2006.

APPENDICES

APPENDIX A

SELECTED SAMPLE APPLICATIONS

This chapter presents the applications' documents with the goal of giving volunteers a place to start to understand the concepts and crosscutting concerns involved, as well as to give some practical advice as to "where to start."

A.1 Gadget Manufacturing System

This is a non-trivial Java enterprise application. It defines a set components of distributed transaction-oriented enterprise application, i.e., *Enterprise Java Bean* (EJB), that contains the business logic operates on the enterprise data, i.e., as *Java Persistence* (JPA). It consists of a set of Java Virtual Machine (JVM) instance that manages distributed transactions, Maven (Project Object Model), Hibernate, Jboss Server, JTA, Arjuna library, and Mysql database. The current EJB architecture supports flat transactions only, but the Arjuna supports nested transactions in the application too.

We used *Java 2 Enterprise Edition* (J2EE) to build this system. J2EE is one of many specifications in the distributed middleware space. When a client with J2EE requests a transaction, the JTA activates the transaction by creating a UserTransaction object, creating an appropriate context, sending an associated data

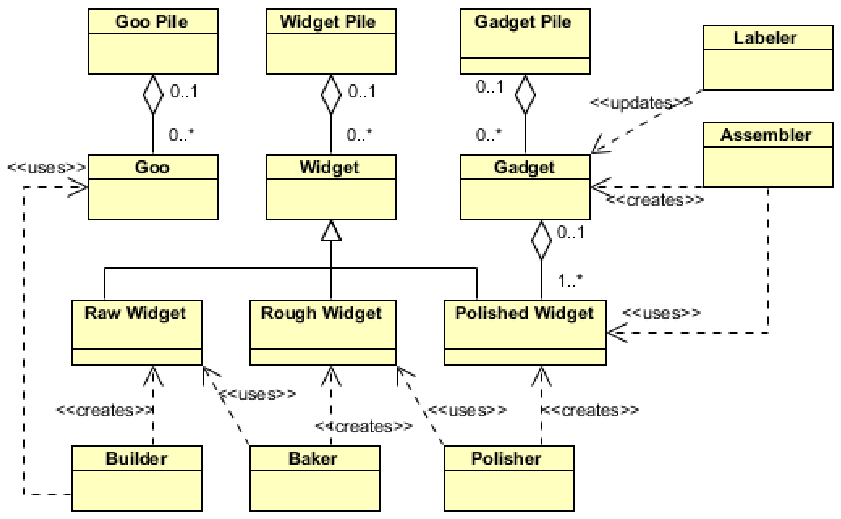


Figure A-1. Resources in a Widget and Gadget Manufacturing System to the appropriate TM (e.g., Builder, Baker, Polisher, Labeler, or Assembler), and then placing the object back in ready state in the instance pool.

Figure A-1 shows an overview of the current architecture for this system. It contains 12 different classes. Pile classes are the distributed shared resources run on separate processes, and separate machines as described in above. Goo, Widget, and Gadget entities provide a remote access to their business logic components, i.e., GooPile, WidgetPile, and GadgetPile, respectively, through the remote interfaces. The interfaces have a collection of transaction operations, which are implemented on the client and server sides (Jboss Server) at deployment time.

The J2EE API provides methods and classes that support these services in the application code. It also can create a UserTransaction object that it associates with the TM. Builder, Baker, Polisher, Assembler, and Labeler are TMs can manage transactions on the piles. They can invoke the transaction object's begin() method at the beginning of the method body, and its commit() or rollback() methods at the end. These transactions are widespread that allow multiple threads to exist within the context of a single host. These threads share the distributed resources, but are able to execute independently with a useful abstraction of concurrent execution, i.e., lock and release abstractions.

The purpose of discussing the design and implementation of this application is to describe the resource sharing data logic in the context of a system that collects data from distributed transactions. As mentioned above (see Chapter 2), this system allows the volunteers to use different types of shared resources (e.g., goo) that are required to build *Widgets* from *Goo* and *Gadgets* from *Widgets*. These resources are distributed on multiple hosts, these hosts can periodically receive requests to produce the desired product.

We imagine the developers want to build a control system to manage the production speed process. Specifically, the system will slow down the Widget production process when the total number of Widgets in the inventory more than 10, otherwise it will speed up the production process, as shown in A-2. This is the UML sequence diagram shows the interaction between different distributed transaction operations that contain the Widget production speed control logic. It is a crosscutting concern spreading and depending on the number of repetitions of the widget and gadget transaction operations. In other words, the rules of Widget production speed control are tangled into the implementation of the *Builder*, *Baker*, *Polisher* and *Assembler*. For a more detailed description of the project, review Chapter 2.

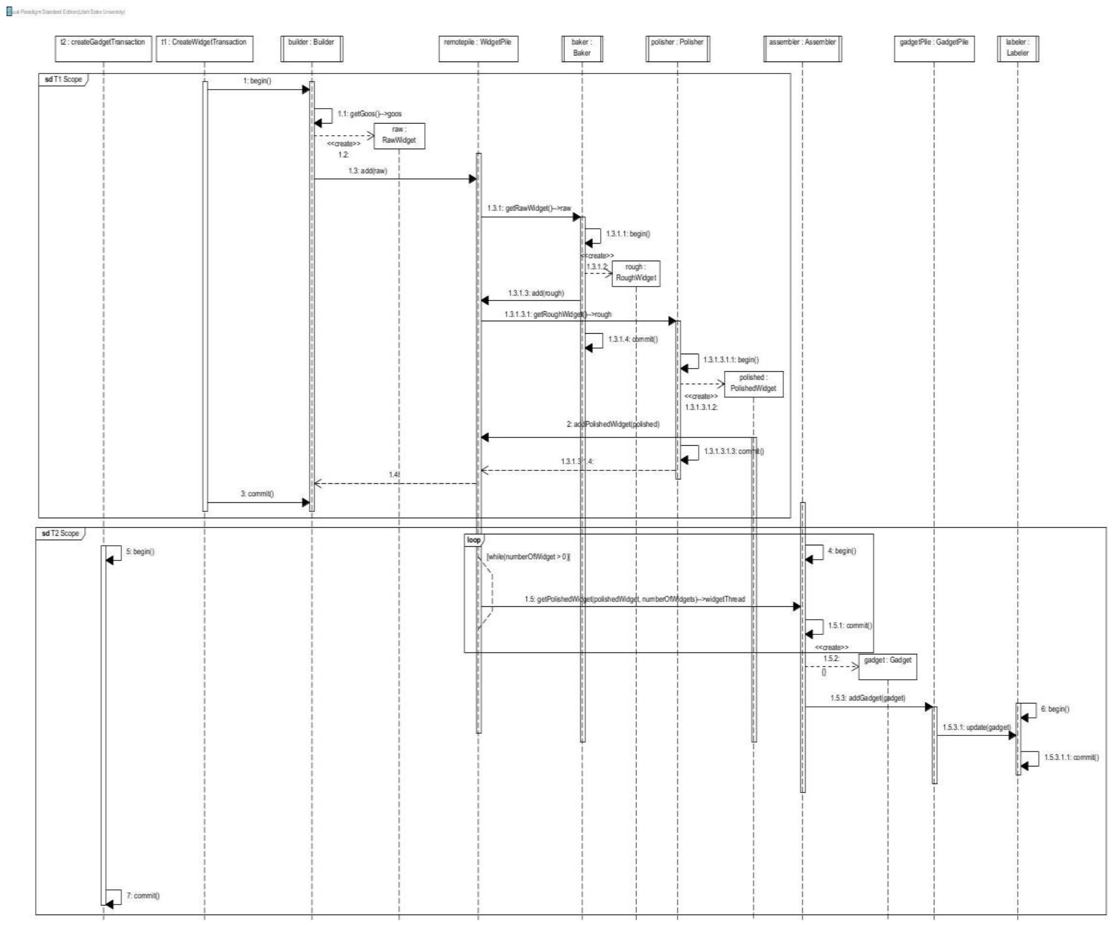


Figure A-2. Interaction Diagram between Distributed Shared Resources

A.2 Bank System

Figure A-3 shows an overview of the architecture for a simple common bank transaction system. This system consists of four simple classes: BankTransactionClient, Bank, AccountManager, and AccountResource.

The BankTransactionClient class encapsulates an interactive command line interface that allows the user of the JBoss Transactions product to manipulate a data resource backed bank transactions under transactional control.

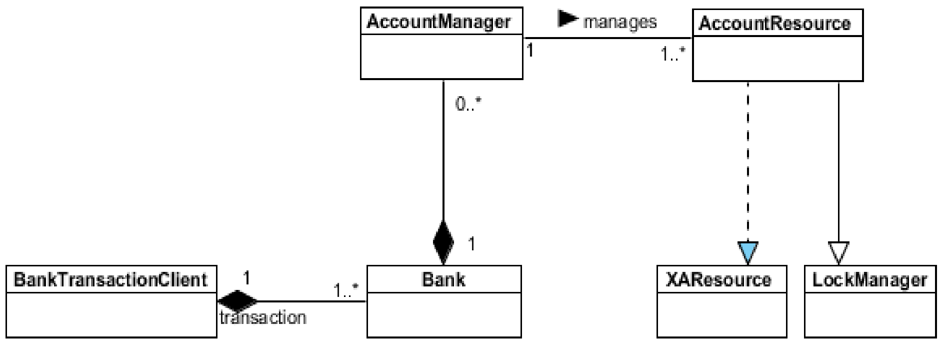


Figure A-3. Architecture Diagram for the Bank Transaction System

The Bank is a data access class that encapsulates the banking transaction operations that all Bank implements in the system, e.g., create an account and get account information. The bank implementation uses the JTA transaction components that can be directly invoked on an AccountManager object rather than multiplexing all calls through the Bank object. The AccountManager reflects a persistent transaction-aware representation of a bank Account resource. As a bank account, it has a balance associated with it. It allows an account balance to be interrogated, increased or decreased, all under transactional control.

Transactionally, the AccountManager manages an AccountResource to update the account balance. The balance of the account is not modified until the transaction is committed or rolled back.

The AccountResource demonstrates a simple bank resource using JTA. It is a very simple implementation of an *XAResource* interface. It supports the reading and writing of a balance under transactional control. Additionally, it extends *LockManager* that provides the standard *TXOJ* capabilities to manage the concurrent activities.

The UML sequence diagram in Figure A-4 shows a transaction is used to make a transfer from a supplier account to consumer account. BankTransactionClient begins the transaction so all work now will be executed under transactional control. The supplier bank sends a debit request to the AccountManager to ask for transfer money from its bank account to the consumer's bank account. Then AccountResource receives the request, it starts updating (i.e., increment) the consumer balance. Then also, customer bank starts updating (i.e., decrement) the supplier's account. Once the transaction has been successfully updated, the transfer

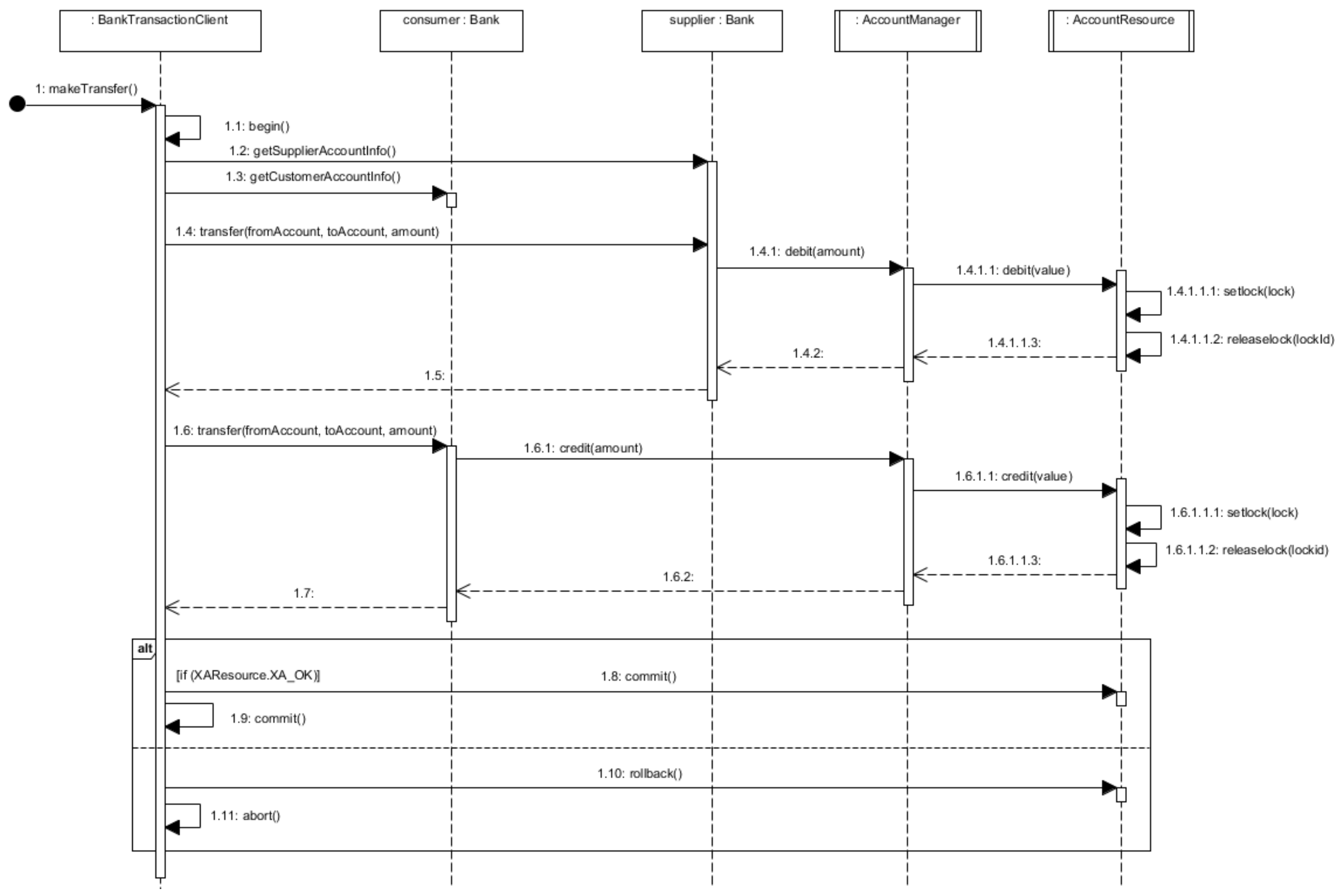


Figure A-4. Transfer Transaction Interactions

transaction commits changes and then clean up itself. However the transfer transaction rollback changes and sends back the denied information if the transaction fails. Both the customer's bank account and supplier's bank account need to be locked until completion the transaction.

A.3 The Conference Registration System (CRS)

The Conference Registration System (CRS) should enable conference attendees to rapidly and easily register for conferences.

The context of this document is to set up preliminary requirements planning, system analysis, system design, and some application design. This system will be able to process transactions for multiple registrations. Each registration will be defined by a unique name. The system tracks registration information and correctly calculate the number of participants based on a registration/de-registration counter. In addition, the CRS tracks multiple transactions are made.

By talking with the domain experts, we would probably discover a couple of basic rules that determine that attendance requirements are not met, this is called “Evaluate Attendance”. In other words, the system checks to see if the amount of registered attendees meets the predefined minimum requirements, e.g., 10 attendees. The number of attendees is not at least this number, the system will cancel the conference. Also, it allows to prevent attendees from inadvertently signing up for twice under the same name to the same conference.

Figure A-5 shows an overview of the current architecture for this system. It consists of various kinds of remote Enterprise Java beans (EJBs): Conference, ConferenceManager, ConferenceManagerEJB, and ConferenceCounter. The Conference is a shared resource that extends the XAResource interface. ConferenceManager defines the basic transaction operations required to add, remove, get information, and list of attendees. ConferenceManagerManagedBean implements the functions of the ConferenceManager. ConferenceManagerEJB shows some transaction business logic. ConferenceManagerEJBImpl implements the logic of transactions that are defined in the ConferenceManagerEJB. ConferenceCounter is a transactional object that behaves like a transaction-aware resource and extends the LockManager functions.

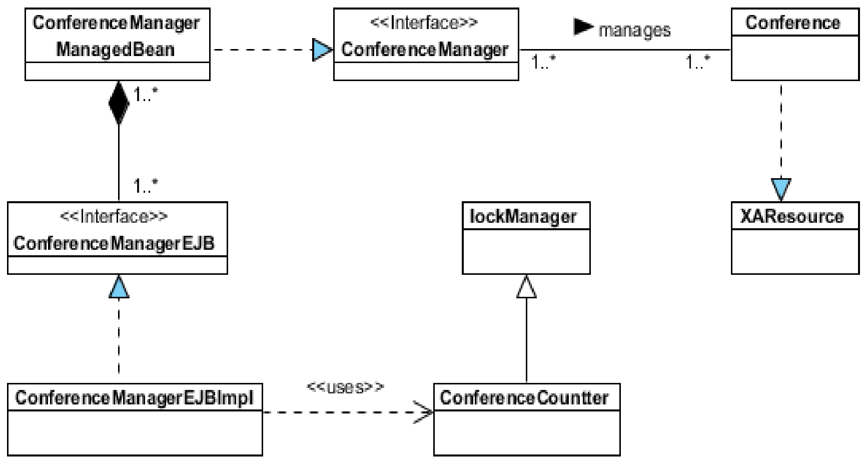


Figure A-5. Architecture of the Conference Registration System

ConferenceManagerBeanImpl uses the ConferenceCounter to control on behalf transactions and managed the concurrent conflicts, as shown in Figure A-6.

Figure A-6 shows a sample of transactions that may occur in the system. A registration transaction is a transaction create a new attendees for a conference. ConferenceManager is a transaction manager begins a new registration transaction to start executing a transaction operation, e.g., add, on the ConferenceManagerEJB, i.e., resource manager, under transactional control. Then, ConferenceCounter starts updating, i.e., increment, the attendance counter. Once the transaction has been successfully updated, the registration transaction commits changes and then clean up itself.

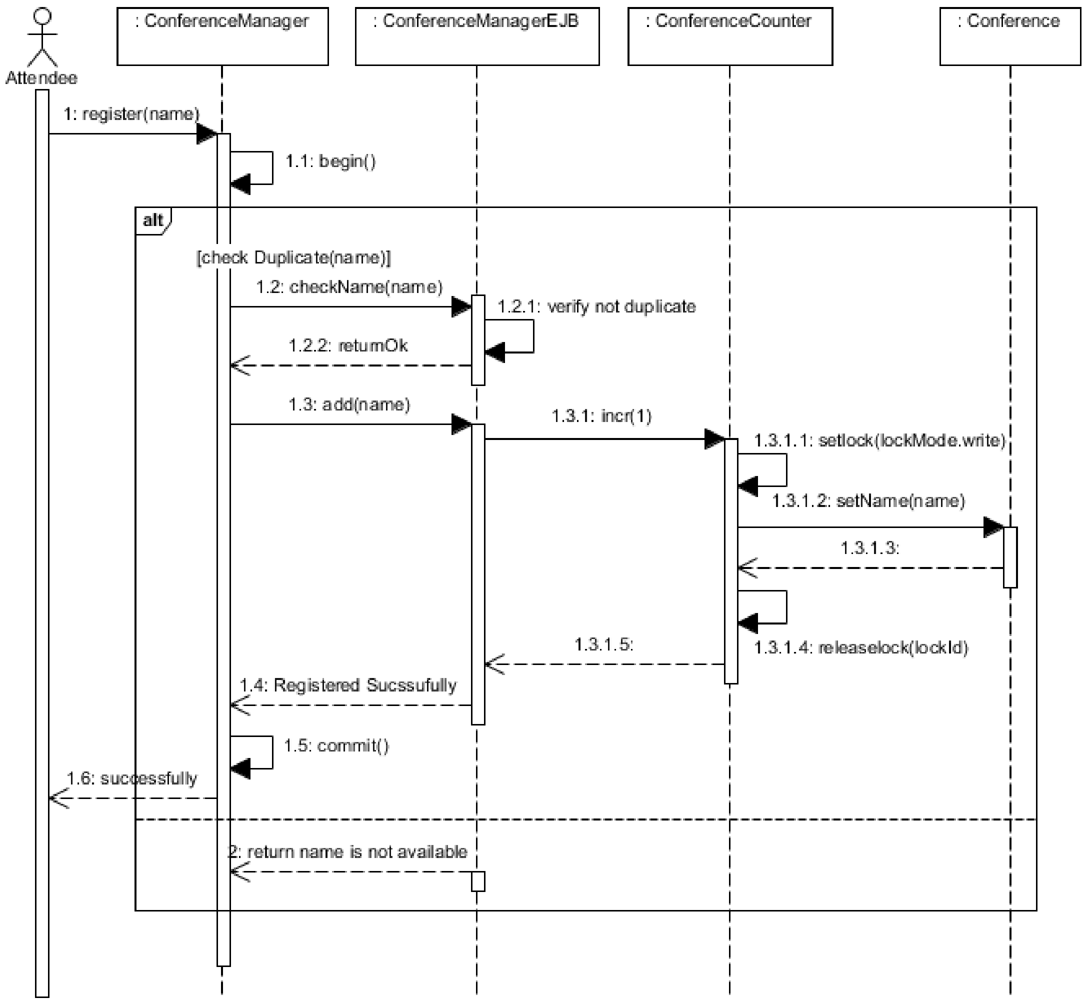


Figure A-6. Registration Transaction Interaction

APPENDIX B

SELECTED TRANSACTION REQUIREMENT EXTENSIONS

B.1 Measuring Performance

This extension measures some performance-related statistics for transactions between a client and application server (JBossAS). In addition, the extension logs the following performance related statistics:

- Total number of transactions, which occurred in the system where a transaction can be defined with any distributed transaction of, begins or commits/aborts. Different type of transaction models is flat transactions or nested transactions.
- Total time for all transactions.
- Total Number of Accesses for all transactions.
- Average turnaround time for a request to be processed where it is the average of a timespan from transaction begin time to transaction commit or abort time.
- Average resource locked time for resources where the average resource locked time is a timespan that spans when a lock is held, starting after acquiring of the lock and ending just before its release.
- The program logs the time when a transaction begins.
- The program logs and calculates the above statistics when the transaction commits or aborts.
- Note that a transaction can be a single transaction or concurrent transaction. We define the transaction for sample applications as follows:
    - Bank System: is a local transaction system consists of computation on local variables, e.g., Balance, with limited to accessing a single resource to take place at a single point in time, i.e., flat transaction. A transaction is when a client begins a request and receives a response from the system.
    - Widget-Gadget Manufacturing System: is a nested distributed transaction system that supports a multithreaded transaction and allows at least two transactions begin a lock request to a shared resource at the same time. In other words, it begins as a request is made and ends when the request is granted or denied. A Transaction when a client requests for

a widget or gadget and when it receives the last confirmation or denied from the parent transaction for that request from the remote host.

    - Conference registration System: is a distributed flat transaction system. A transaction is when a user requests for registering or de-registering until he/she get confirmation or denied.

- The developer provided with the following classes:
    - Reusable Aspects: Aspects containing elements to measure performance.
    - Performance Measure: It records performance measures (Transaction Volumes and Velocity) using logs.

B.2 Audit Trail

The audit trail is a time-stamped record of significant activities and actions on a system, show system threads of access, modifications, and transactions. In CRS, recorded events can include user registration and deregistration in the conference, as well as what names were issued by the attendees to the register for the conference. In other words, an audit trail is scattered through the code to keep track of who registered, and when they registered, as well as who tried to do registration but was unsuccessful. It's useful for detecting conflicted registration, establishing a culture of responsibility and accountability, decreasing the risk associated with inappropriate registrations, detecting new threats and interruption attempts, and recognizing possible problems, etc.

B.3 Data Sharing Optimization

In order to increase throughput and hence maximize performance of the GMS, Shared needs the context information on each operation of the transaction provided by operations that read and write the data of resources in many shared contexts. The GMS allows the threads running concurrently within the same transaction may simultaneously execute conflicting transaction operations on a shared resource — producing an invalid or inconsistent state. It is therefore necessary to prevent threads from concurrently modifying a transaction’s state. Such a situation could arise when the operations of the same transaction want to concurrently access the same resource. The developer needs to develop a reusable or an application level aspect, e.g., *Shared Aspect*, which provides this functionality (see Chapter 6). It provides mutual exclusive access of a transactional resource to either a single context operation or context of multiple concurrent read operations.

APPENDIX C

SKILL ASSESSMENT SURVEY

Volunteer ID ____________

1. How many years of programming experience do you have?

   a. No prior programming experience
   b. Less than 1 year
   c. Between 1-3 years
   d. Between 3-5 years
   e. More than 5 years

For each CONCEPT listed below, rate your knowledge level on a scale of 1 to 5 -- 1=Low and 5=High.

2. Novices will have a working knowledge of the skill, but no practical experience. An intermediate will have at least 2 years of practical experience, in either academic or industrial settings. An expert will have more than 3 years of experience.

a.	Java programming language.	1 2 3 4 5
b.	Aspect-Oriented Programming, Like AspectJ	1 2 3 4 5
c.	Unified Modeling Language (UML).	1 2 3 4 5
d.	Good design principles such as modularity etc.	1 2 3 4 5
e.	Multithreaded programming using Java.	1 2 3 4 5
f.	Java Transaction API (JTA).	1 2 3 4 5
g.	Transaction Manager (TM) such as JbossTS, Bitronix, Atomikos, or SpringTM	1 2 3 4 5
h.	Enterprise Application Platform (EAP), Jboss Application Server, or WildFly.	1 2 3 4 5
i.	Javax.transaction library.	1 2 3 4 5
j.	Concurrency control programming.	1 2 3 4 5
k.	Distributed programming application.	1 2 3 4 5
l.	Java Database Connection (JDBC.)	1 2 3 4 5
m.	Database Management Systems (MsSql, MySql, postgress, DB2, or other.)	1 2 3 4 5
n.	Enterprise Java bean (EJB)	12 3 4 5
o.	Java Persistence Application (JPA)	1 2 3 4 5
p.	Apache Maven	1 2 3 4 5

3. Please quantify your most complex Java programming project in terms of *Lines of Code* (LoC)?

   a. Less than 1,000 LoC
   b. Between 1,000 and 10,000 LoC
   c. Between 10,000 and 20,000 LoC
   d. Between 20,000 and 100,000 LoC
   e. More than 100,000 LoC

4. How many years of *Java* programming experience do you have?

a. No prior programming experience
b. Less than 1 year
c. Between 1-3 years
d. Between 3-5 years
e. More than 5 years

5. Please select your favorite programming languages?

a. Java
b. C#
c. PHP
d. Ruby and Rails
e. Python
f. C++
g. Other

6. How many years of *Aspect-oriented* programming experience do you have?

a. No prior programming experience
b. Less than 1 year
c. Between 1-3 years
d. Between 3-5 years
e. More than 5 years

7. Please select your favorite Aspect-Oriented Software Design (AOSD) (circle all that apply)?

a. AspectJ
b. SpringAOP
c. JbossAOP
d. C++ Aspect
e. Other (___________________)

8. What is your computer science education background?

a. BS/BE
b. MS
c. Ph.D.
d. Other (___________________)

9. Which of the following courses have you taken as part of your computer science curricula? (circle all that apply)

a. Object Oriented Design and Programming
b. Software Engineering
c. Unified Modeling Language
d. Multithreaded Programming
e. Distributed Systems
f. Database Systems
g. Concurrency Control Systems
h. Aspect-Oriented Programming

APPENDIX D

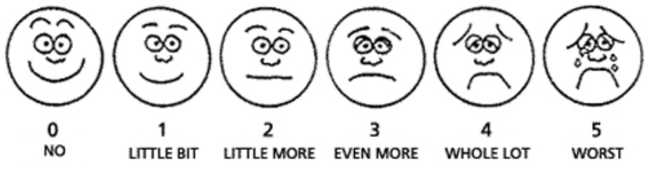
QUESTIONNAIRE FOR IMPLEMENTATION

Volunteer # ________________

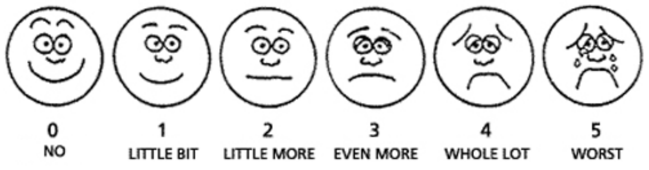
Application ______________________________

D.1 Pre-implementation questionnaire

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



2. On a scale from 0-5, how would you rank the existing applications for code scattering (5 means fully scattered in all classes and 0 means no scattering)?



3. Now if you were asked to change the implementation for (*Gadget Manufacturing System,*) to implement Control Speed Rules, how would your changes have been? (circle one)
   a. Considerably different
   b. Somewhat different
   c. A little different
   d. No different

4. Now if you were asked to change the implementation for (*Conference Registration System*) to implement Audit Trail Tracker, how would your changes have been?
   a. Considerably different
   b. Somewhat different
   c. A little different
   d. No different

5. Now if you were asked to change the implementation for (*Bank System*) to calculate turnaround time, how would your changes have been?

   a. Considerably different
   b. Somewhat different
   c. A little different
   d. No different

6. If the original application of *(Gadget Manufacturing System*) were implemented in such a way so that the transactions in the original application can be nested distributed transactions, would your changes be:

   a. Considerably different
   b. Somewhat different
   c. A little different
   d. No different

7. Suppose we want to implement the “Performance Measurement” feature for the original applications. The feature measures some performance related statistics such as turn-around time for transactions between a client and host. To implement this feature, your changes would be?

   a. Considerably different
   b. Somewhat different
   c. A little different
   d. No different

8. Now suppose if we change the requirements for the “Performance Measurement” feature such as turn-around time for nested distributed transactions between multiple clients and hosts, would this change be:

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

9. Now suppose if we change the requirements for the “Audi Trail” feature such as a log for nested distributed transactions between multiple clients and hosts, would this change be?

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

10. Now suppose if we change the requirements for the “Shared Resources” feature such as add more shared resources for nested distributed transactions between multiple clients and hosts, would this change be:

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

D.2 Post-implementation questionnaire

Volunteer # ____________

Application ______________________________

1. While implementing the initial version of changes to the application, which of the following did you find the most difficult? (circle all that apply)

    a. Adding additional requirements for the extension part to applications design
    b. Deciding how to share data between previously existing sample application code and new code
    c. Debugging the applications with crosscutting concerns
    d. Working with the Java implementation language or the IDE
    e. Managing the complexity of the application

2. Which of the following was the most time consuming activity during implementation? (circle one)

    a. Understanding the original applications and analyzing the new requirements
    b. Designing the solutions
    c. Implementing the solutions
    d. Debugging the solutions
    e. Learning the tools (e.g., Java, an IDE, JTA, JBoss)
    f. Learning AOP
    g. Learning *TransJ* (not applicable to Group 1)

3. While implementing your changes, did you come across any of the following situations? (Circle all that apply)

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

4. If you were asked to refactor the changes related to the extension part so it could be reused by other applications, which of the following would you do?

    a. Redesign the application's structure, making major changes in the classes, their relationships, and responsibilities
    b. Refactor the code to make minor improvements to the classes, their relationships, or responsibilities
    c. Improve the implementation of individual methods, independent of changing the structure of the application, to improve reusability, readability without scarifying in performance.
    d. Nothing - the implementation is ready for reuse

5. How would you rank your application, so that it would work again if you separate the extension related code files from the sample application code?

    a. Very easy to change, the two parts are almost oblivious
    b. A little difficult, as there are some extension-related references in the original application
    c. A significant effort is required, as some extension-related code snippets are tangled and scattered in the original application code or vice-versa.

6. Suppose your original application (Gadget Manufacturing System) were implemented using another Transaction Manager (such as jtaTransactionManager in Spring or Bironix). To implement this feature, your changes would be:

    a. Considerably different
    b. Somewhat different
    c. A little different
    d. No different

7. If the original application of (Bank system) were implemented in such a way that the *Transaction*s in the original application shared multiple resources when necessary (Just In time), to implement this feature your changes would be:

    a. Considerably different
    b. Somewhat different
    c. A little different
    d. No different

8. If the original applications were implemented using Bitronix's Transaction Manager rather than Jboss's Transaction Manager, to implement this feature, would your changes are:

    a. Considerably different
    b. Somewhat different
    c. A little different
    d. No different

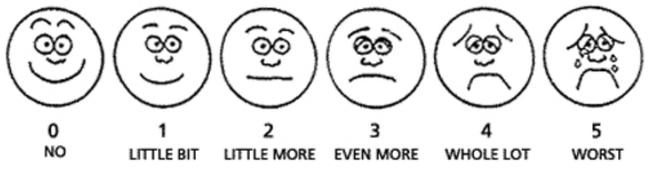
9. To implement the “Performance Measurement”, “Audit Trail, or “Optimization shared resources” feature, which of the following changes did you make in your original application?

    a. Need to introduce major changes in the original application code
    b. Need to introduce new pointcuts
    c. Need to define new data structures to keep track of transactions
    b. Lines of Code (LoC) and complexity of sample application may increase
    c. Tangling and Scattering of sample application may increase
    d. Require only minor change in implementation
    e. May expect some new bugs in the program
    f. Overall debugging time would dramatically increase
    g. Can reuse existing code to implement the new changes

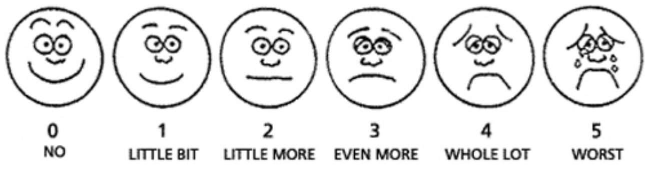
10. Suppose if we change the requirements for “Performance Measurement” feature such that a transaction is not only a flat transaction, but also a nested (multiple remote hosts and multiple threads), what are the following changes you can expect in your implementation and efficiency?

    a. Need to introduce major changes in the original application code
    b. Need to introduce new pointcuts
    c. Need to define new data structures to keep track of transactions
    d. Lines of Code (LoC) and complexity of sample application may increase
    e. Tangling and Scattering of sample application may increase
    f. Degrading in the performance
    g. Require only minor change in implementation
    h. May expect some new bugs in the program
    i. Overall debugging time would dramatically increase
    j. Can reuse existing code to implement the new changes

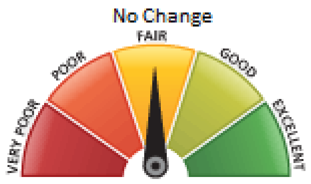
11. On a scale from 0-5, how would you rank the overall application after the changes you implemented in for code tangling (5 means fully tangled and 0 means two are totally independent)?



12. On a scale from 0-5, how would you rank the overall application after the changes you implemented in for code scattering (5 means fully scattered in all classes and 0 means no scattering)?



13. On a scale from very poor- excellent, how would you rank the overall application performance after the changes you implemented?



14. How many hours did you spend reading and understand the code?

15. How many bugs and errors you've got during development?

16. How many hours did you spend to trace and correct bugs and errors?

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

    a. Performance measurement:
    b. Audit trail:
    c. Data-sharing optimization:
    d. Turnaround time:

18. If you want to use a sequential-transaction model instead of a concurrent-transaction model in the Gadget - manufacturing system, you need to make the following code modifications?

    a. No change in implementation was required
    b. Need major changes such as creating new classes
    c. Need moderate changes such as creating new methods and variables
    d. Need minor changes such as modifying few existing methods and variables
    e. None of the above

19. What of the following was the most time consuming during implementation of feature changes?

    a. Understanding the original applications and analyze the new requirements
    b. Designing the solutions
    c. Implementing the solutions
    d. Debugging the solutions
    e. Learning the tools (e.g., Java, an IDE, EJB, JPA, JTA, JBossAS, Arjuna)
    f. Learning AOP
    g. Learning *TransJ* (not applicable group 1)

20. If your original application was implemented using another application server, such as Tomcat, Spring or Atomatiks. To implement this modification you made?

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

21. If your original application of (Bank System) was implemented using nested Transactions. To implement this modification you made?

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

22. Would your application be able to run standalone again if you remove the feature changes from sample application code?

    a. Yes
    b. No
    c. Not sure

23. In order to implement the change in requirements for "Performance Measurement" feature such that a transaction is not only on a single resource, but also a distributed shared-resources (multiple resource hosts), what are the following changes you made in your implementation?

    a. Need to introduce major changes in the original application code
    b. Need to introduce new pointcuts
    c. Need to define new data structures to keep track of transactions
    d. Lines of Code (LoC) and complexity of sample application may increase
    e. Tangling and Scattering of sample application may increase
    f. Require only minor change in implementation
    g. May expect some new bugs in the program
    h. Overall debugging time would dramatically increase
    i. Can reuse existing code to implement the new changes
    j. None of the above

APPENDIX E

DATA ASSESSMENT FROM THE SURVEYS

To assess volunteers’ skill levels, we designed a skills assessment survey (see Appendix C) to identify individual developers’ academic and technology skill levels and then create a starting point for developing application requirements. The observations we obtained from this survey support our initial requirements about the selection and background of the participants in the experiment mentioned in Chapter 9.

E.1 Programming Experience

Figure E-1 reveals that all the participants bring a variety of programming experience levels and educational backgrounds to bear on our experiment. The participants have a programming experience ranging from moderate-level to expert-level in programming. From the graph in Figure E-1, we can see that 25% of participants had 1-3 years of experience, 50% of participants had 3-5 years of experience, and 25% of participants had over 5 years of experience.

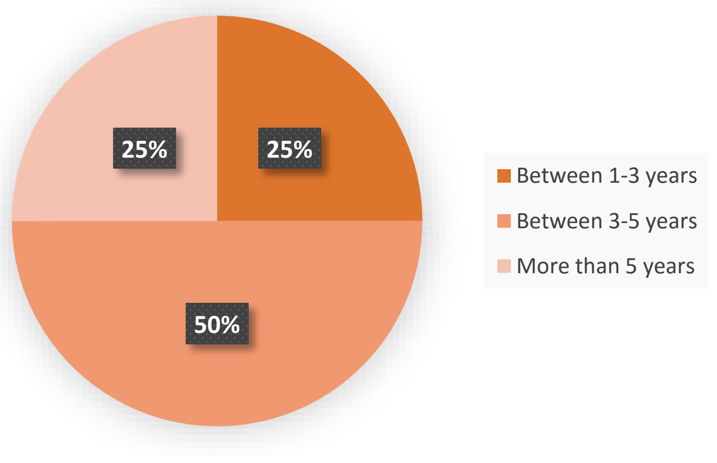


Figure E-1. Programming Experience of Selected Volunteers

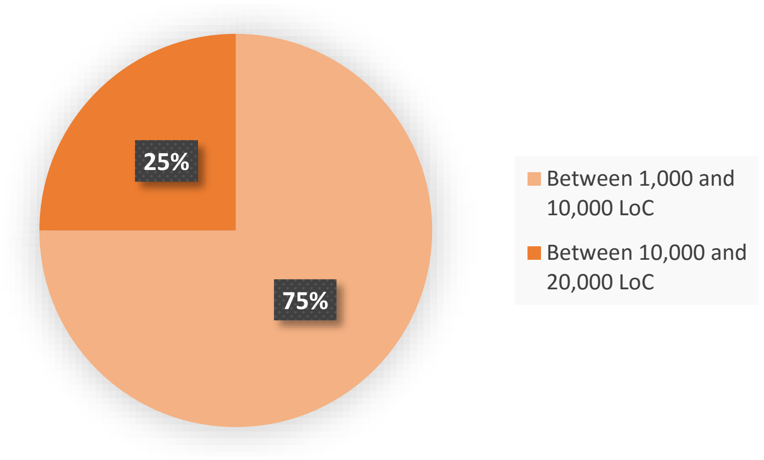


Figure E-2. Previous Projects LOC of the Selected Volunteers

The graph in Figure E-2 shows 75% of participants had experience in developing programs up to 1000-10000 LOC and 25% of participants had experience in developing programs up to 10000- 20000 LOC.

E.2 Programming Language and Software Engineering-Specific Skill Set

Figure E-3 shows that the following observations about the participant skills.

- Almost 75% of the participants had advanced-level expertise in understanding.
- Level of familiarity and expertise in AspectJ
    - Almost 25% of the participants had basic-level expertise
    - Almost 50% of the participants had intermediate-level expertise
    - Almost 25% of the participants had advanced-level expertise
- Collectively, 100% of the participants were found to have an advance or expert level in understanding and applying the UML.
- 100% of the participants had advanced-level expertise in understanding and applying good design principles.
- Level of expertise in transaction-related programming concepts

- Almost 50% of participants had intermediate-level expertise in applying multithreaded, concurrency technique, distributed, JDBC, Database Management Systems, EJB, JPA and Maven concepts. Our tutorial on these programming concepts proved helpful for the participants to comfortably implement the required programming tasks in the experiment.
- Almost 75% of participants had basic-level expertise in Transaction Manager Technology. Our tutorial on these transaction manager implementations proved helpful for the participants to comfortably use the required Arjuna transaction manager in the experiment.
- Almost 50% of participants had basic-level expertise in Enterprise Application Platform (EAP), Wildfly, or Jboss Application Server. Our tutorial on these application servers proved helpful for the participants to comfortably use the required server in the experiment.

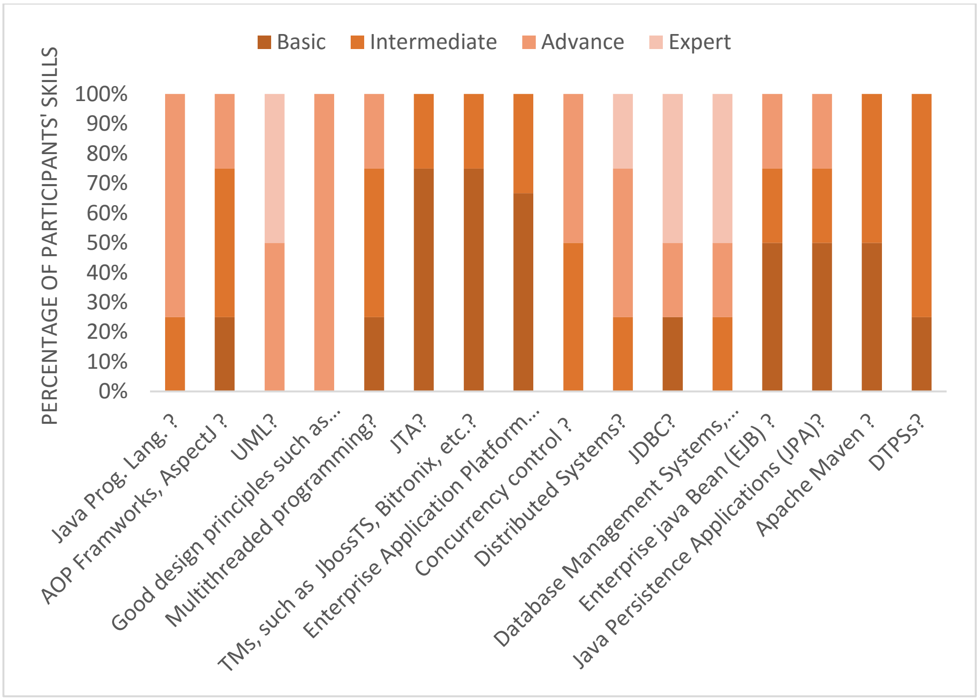


Figure E-3. Specific Skills Set of the Selected Volunteers

- Almost 75% of participants had intermediate-level expertise in applying DTPS concepts. Our tutorial on the distributed transaction concepts proved helpful for the participants to comfortably use transaction-related concepts in the experiment.
- Almost 75% of the participants had basic or no familiarity with transaction programming in JTA in Java. Hence, we arranged tutorials on transaction programming, and in later surveys, participants described themselves as having a sufficient understanding to implement the transaction-related concepts in the experiment.

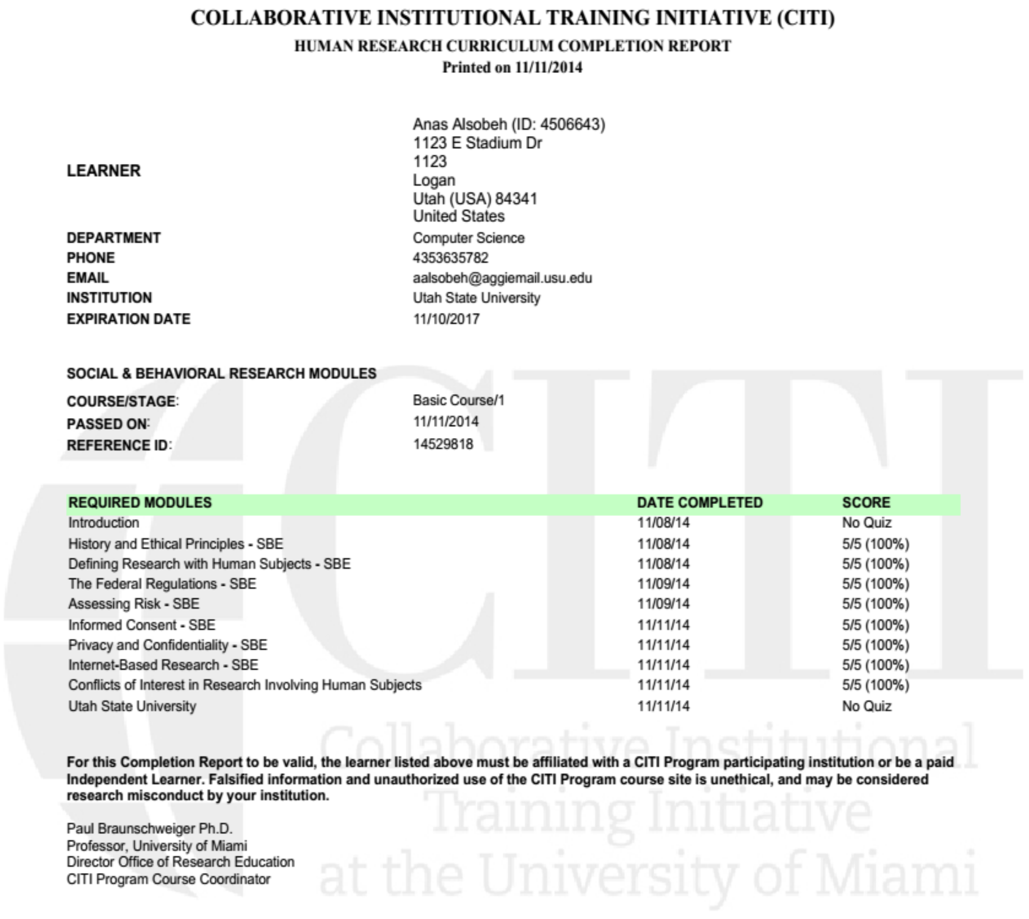
We concluded from the data in the above graphs that the participants shared a common background in OO concepts, previous programming experience, and level of projects completed in the past, as well as understanding and applying good software engineering principles. Hence, the selected participants were found to sufficiently fulfill the requirements related to the selection of the developers in chapter 9.

APPENDIX F

DOCUMENTS FOR THE RESEARCH EXPERIMENT APPROVAL

F.1 Collaborative Institutional Training Initiative (CITI) Passing Report

As per requirement of IRB, the student researcher should be became CITI certified prior to engaging in any research-related activities. In doing so, I have passed the Human Subject Research course in order to meet the pressing need for human subjects protections. Fig shows the detail of the CITI passing report.



COLLABORATIVE INSTITUTIONAL TRAINING INITIATIVE (CITI)
HUMAN RESEARCH CURRICULUM COMPLETION REPORT
Printed on 11/11/2014

LEARNER	Anas Alsobeh (ID: 4506643) 1123 E Stadium Dr 1123 Logan Utah (USA) 84341 United States
DEPARTMENT	Computer Science
PHONE	4353635782
EMAIL	aalsobeh@aggiemail.usu.edu
INSTITUTION	Utah State University
EXPIRATION DATE	11/10/2017

SOCIAL & BEHAVIORAL RESEARCH MODULES

COURSE/STAGE:	Basic Course/1
PASSED ON:	11/11/2014
REFERENCE ID:	14529818

REQUIRED MODULES	DATE COMPLETED	SCORE
Introduction	11/08/14	No Quiz
History and Ethical Principles - SBE	11/08/14	5/5 (100%)
Defining Research with Human Subjects - SBE	11/08/14	5/5 (100%)
The Federal Regulations - SBE	11/09/14	5/5 (100%)
Assessing Risk - SBE	11/09/14	5/5 (100%)
Informed Consent - SBE	11/11/14	5/5 (100%)
Privacy and Confidentiality - SBE	11/11/14	5/5 (100%)
Internet-Based Research - SBE	11/11/14	5/5 (100%)
Conflicts of Interest in Research Involving Human Subjects	11/11/14	5/5 (100%)
Utah State University	11/11/14	No Quiz

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

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

Figure F-1. CITI Passing Report

F.2 Research Experiment Invitation Letter

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

LETTER OF INVITATION TO PARTICIPATE IN A RESEARCH STUDY

Investigation into the Benefits of Weaving Aspects into Distributed Transactions

Date: 07/15/2015

Dear Students,

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

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

- You are enrolled for a degree program in Computer Science
- You have good exposure of object-oriented and Unified Modeling Language (UML)
- You have taken at least one programming course in Java
- You have taken at least one software-engineering class
- You have exposure to database and transaction concepts in Java
- You have exposure to distributed systems.
- (Optional) You have exposure to at least one of application servers, such as Jboss, Tomcat, etc.

By helping us in our research study, you are contributing to the advancement of software engineering tools and methods for transaction applications. In addition to receiving a $300 stipend, you may also receive the following benefits by participating in the study:

- New skills in Aspect-Oriented Programming (AOP).
- An opportunity to learn a new software development framework, namely *TransJ*.
- Additional practice and experience with object-oriented design aspect-oriented design, and software engineering principles.

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

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

We look forward to your participation. If you have any questions about the experiment or your role, please contact Dr. Stephen Clyde (PI) at (435) 797-2307 or Stephen.Clyde@usu.edu and Anas AlSobeh (student researcher) at (435) 363-5782 or aalsobeh@aggiemail.usu.edu.

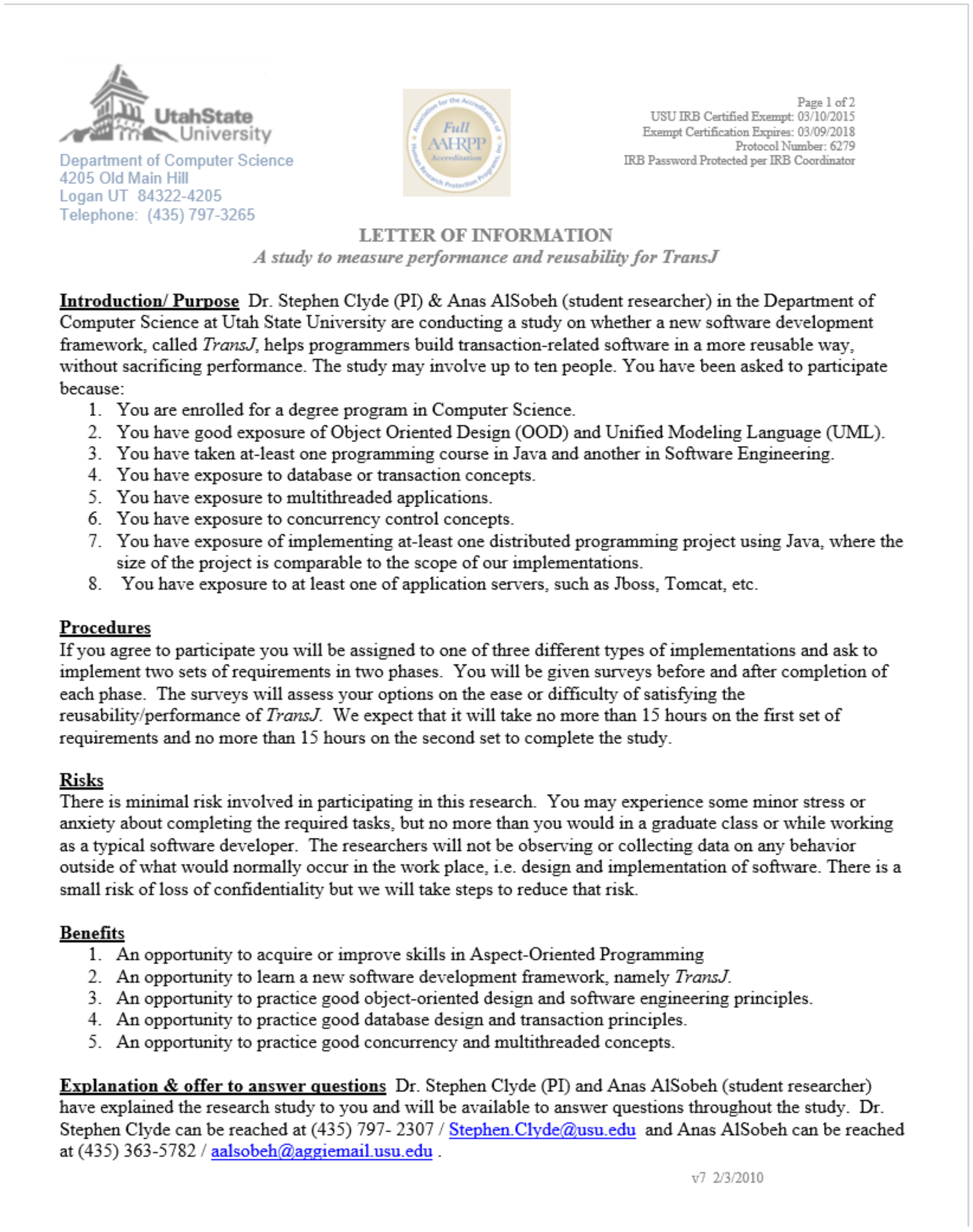
Regards,

Dr. Stephen Clyde (Principal Investigator)

Anas AlSobeh (Student Researcher)

F.3 IRB Approval Letter

IRB evaluated and approved the research experiment application. The approval letter is shown in Figure F-2 below.



UtahState University

Department of Computer Science
4205 Old Main Hill
Logan UT 84322-4205
Telephone: (435) 797-3265

Full AAHRPP Accreditation

Page 1 of 2
USU IRB Certified Exempt: 03/10/2015
Exempt Certification Expires: 03/09/2018
Protocol Number: 6279
IRB Password Protected per IRB Coordinator

LETTER OF INFORMATION

A study to measure performance and reusability for TransJ

Introduction/ Purpose Dr. Stephen Clyde (PI) & Anas AlSobeh (student researcher) in the Department of Computer Science at Utah State University are conducting a study on whether a new software development framework, called *TransJ*, helps programmers build transaction-related software in a more reusable way, without sacrificing performance. The study may involve up to ten people. You have been asked to participate because:

1. You are enrolled for a degree program in Computer Science.
2. You have good exposure of Object Oriented Design (OOD) and Unified Modeling Language (UML).
3. You have taken at-least one programming course in Java and another in Software Engineering.
4. You have exposure to database or transaction concepts.
5. You have exposure to multithreaded applications.
6. You have exposure to concurrency control concepts.
7. You have exposure of implementing at-least one distributed programming project using Java, where the size of the project is comparable to the scope of our implementations.
8. You have exposure to at least one of application servers, such as Jboss, Tomcat, etc.

Procedures

If you agree to participate you will be assigned to one of three different types of implementations and ask to implement two sets of requirements in two phases. You will be given surveys before and after completion of each phase. The surveys will assess your options on the ease or difficulty of satisfying the reusability/performance of *TransJ*. We expect that it will take no more than 15 hours on the first set of requirements and no more than 15 hours on the second set to complete the study.

Risks

There is minimal risk involved in participating in this research. You may experience some minor stress or anxiety about completing the required tasks, but no more than you would in a graduate class or while working as a typical software developer. The researchers will not be observing or collecting data on any behavior outside of what would normally occur in the work place, i.e. design and implementation of software. There is a small risk of loss of confidentiality but we will take steps to reduce that risk.

Benefits

1. An opportunity to acquire or improve skills in Aspect-Oriented Programming
2. An opportunity to learn a new software development framework, namely *TransJ*.
3. An opportunity to practice good object-oriented design and software engineering principles.
4. An opportunity to practice good database design and transaction principles.
5. An opportunity to practice good concurrency and multithreaded concepts.

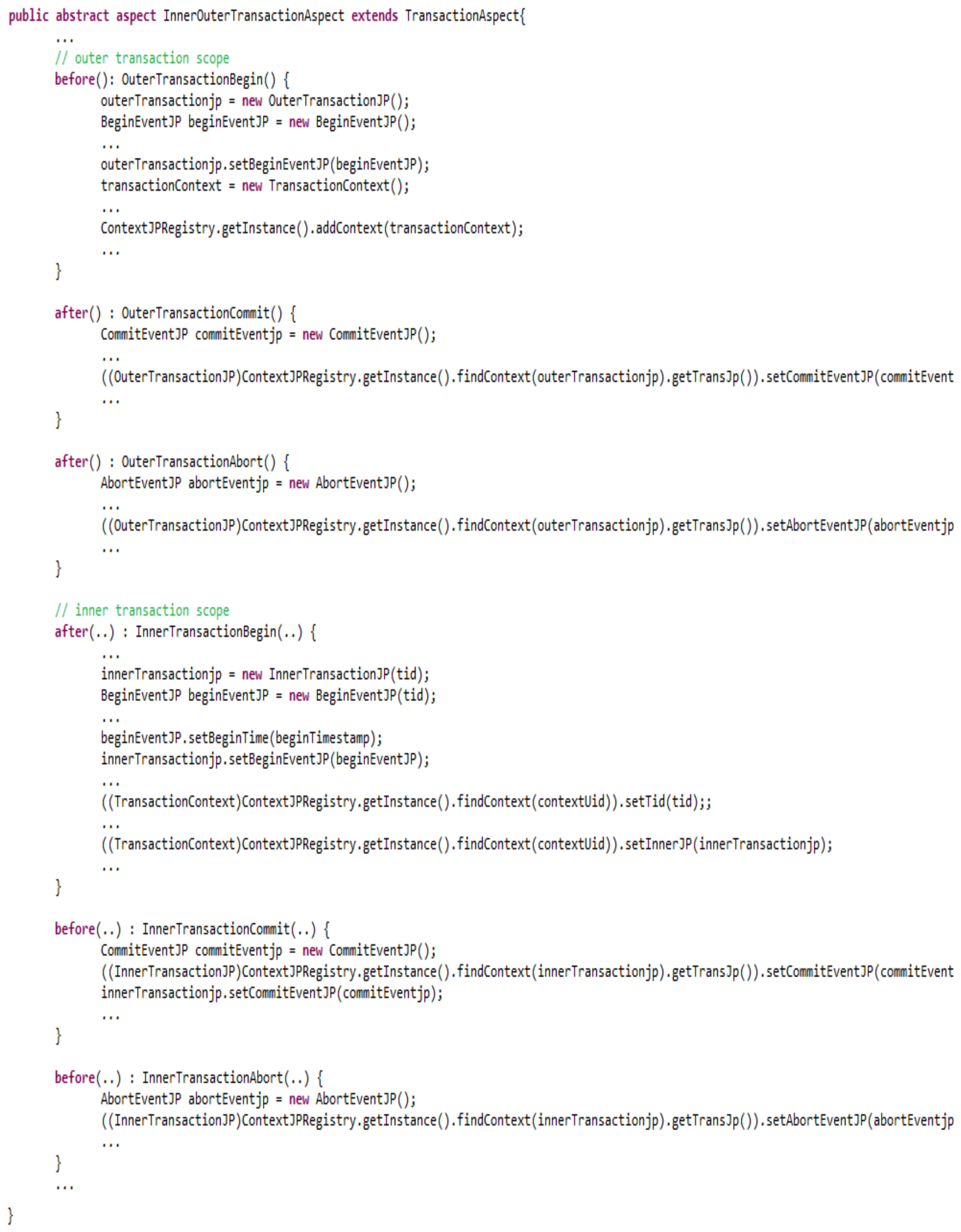
Explanation & offer to answer questions Dr. Stephen Clyde (PI) and Anas AlSobeh (student researcher) have explained the research study to you and will be available to answer questions throughout the study. Dr. Stephen Clyde can be reached at (435) 797- 2307 / Stephen.Clyde@usu.edu and Anas AlSobeh can be reached at (435) 363-5782 / aalsobeh@aggiemail.usu.edu .

v7 2/3/2010

Figure F-2. IRB Approval Letter

APPENDIX G

ADDITIONAL ASPECT CODE SNIPPETS OF TRANSJ IMPLEMENTATION



```
public abstract aspect InnerOuterTransactionAspect extends TransactionAspect{
	...
	// outer transaction scope
	before(): OuterTransactionBegin() {
		outerTransactionjp = new OuterTransactionJP();
		BeginEventJP beginEventJP = new BeginEventJP();
		...
		outerTransactionjp.setBeginEventJP(beginEventJP);
		transactionContext = new TransactionContext();
		...
		ContextJPRegistry.getInstance().addContext(transactionContext);
		...
	}

	after() : OuterTransactionCommit() {
		CommitEventJP commitEventjp = new CommitEventJP();
		...
		((OuterTransactionJP)ContextJPRegistry.getInstance().findContext(outerTransactionjp).getTransJp()).setCommitEventJP(commitEvent
		...
	}

	after() : OuterTransactionAbort() {
		AbortEventJP abortEventjp = new AbortEventJP();
		...
		((OuterTransactionJP)ContextJPRegistry.getInstance().findContext(outerTransactionjp).getTransJp()).setAbortEventJP(abortEventjp
		...
	}

	// inner transaction scope
	after(..) : InnerTransactionBegin(..) {
		...
		innerTransactionjp = new InnerTransactionJP(tid);
		BeginEventJP beginEventJP = new BeginEventJP(tid);
		...
		beginEventJP.setBeginTime(beginTimestamp);
		innerTransactionjp.setBeginEventJP(beginEventJP);
		...
		((TransactionContext)ContextJPRegistry.getInstance().findContext(contextUid)).setTid(tid);;
		...
		((TransactionContext)ContextJPRegistry.getInstance().findContext(contextUid)).setInnerJP(innerTransactionjp);
		...
	}

	before(..) : InnerTransactionCommit(..) {
		CommitEventJP commitEventjp = new CommitEventJP();
		((InnerTransactionJP)ContextJPRegistry.getInstance().findContext(innerTransactionjp).getTransJp()).setCommitEventJP(commitEvent
		innerTransactionjp.setCommitEventJP(commitEventjp);
		...
	}

	before(..) : InnerTransactionAbort(..) {
		AbortEventJP abortEventjp = new AbortEventJP();
		((InnerTransactionJP)ContextJPRegistry.getInstance().findContext(innerTransactionjp).getTransJp()).setAbortEventJP(abortEventjp
		...
	}
	...
}
```

Figure G-1. A Code Snippet of InnerOuterTransactionAspect

```
public abstract aspect TransactionOperationAspect extends TransactionAspect{
     ...
     before(..) : BeforeOperation(..) {

          operationjp = new OperationJP();
          ...
          BeforeOperationEventJP beforeEventJP = new BeforeOperationEventJP();
          ...
          operationjp.setBeforeOperationEventjp(beforeEventJP);
          ...
          operationContext = new OperationContext();
          operationContext.setOperationjp(operationjp);
          ...
          ContextJPRegistry.getInstance().addContext(operationContext);
     }

     after(..) : AfterOperation(..){

          AfterOperationEventJP afterEventJP = new AfterOperationEventJP();
          ...
          ((OperationJP)
ContextJPRegistry.getInstance().findContext(operationjp).getTransJp()).setAfterOperationEventjp(afterEventJP);
          ...
          operationjp.setAfterOperationEventjp(afterEventJP);
     }
     ...
}
```

Figure G-2. A Code Snippet of TransactionOperationAspect

```
public abstract aspect LockAspect extends TransactionAspect {

          ...

          before(..) : BeginlockOperation(..) {
               lockingjp = new LockingJP();
               BeginLockEventJP beginlockEventjp = new BeginLockEventJP();
               ...
               lockingjp.setBeginlockEventjp(beginlockEventjp);
               ...
               lockContext = new LockContext();
               ...
               lockContext.setLockingjp(lockingjp);
               ContextJPRegistry.getInstance().addContext(lockContext);
          }

          after (..) : EndlockOperation(..) {
               EndLockEventJP endlockEventjp = new EndLockEventJP();
               ...
               endlockEventjp.setElapsedTime(elapsedTime);
               ...
               ((LockingJP)
ContextJPRegistry.getInstance().findContext(lockingjp).getTransJp()).setEndlockEven
tjp(endlockEventjp);

               lockingjp.setEndlockEventjp(endlockEventjp);
               ...
          }

          after(..): SetHold(..) {
               resourceLockedjp = new ResourceLockedJP();
               HoldEventJP holdEventJP = new HoldEventJP(lockOwnerId);
               ...
               resourceLockedjp.setHoldEventjp(holdEventJP);
               ...
               ((LockContext)
ContextJPRegistry.getInstance().findContext(lockingjp)).setResourceLockedjp(resourc
eLockedjp);
               lockContext.setResourceLockedjp(resourceLockedjp);
               ...
          }

          before(..) : ReleaseHold(..) {
               ...
               ReleaseEventJP releaseEventjp = new ReleaseEventJP();
               ...
               ((ResourceLockedJP)
ContextJPRegistry.getInstance().findContext(resourceLockedjp).getTransJp()).setRele
aseEventjp(releaseEventjp);
               resourceLockedjp.setReleaseEventjp(releaseEventjp);
               ...
          }
```

Figure G-3. An Aspect Code Snippet of LockAspect

APPENDIX H

CURRICULUM VITAE

Anas "Mohammad Ramadan" Ahmed AlSobeh
Utah State University –Logan, UT 84321
Engineering Collage
Computer Science Department
Ph.D. Computer Science
E-mail: anas.alsobeh@usu.edu,
aalsobeh@yahoo.com,
Mobile: (+1) 435-363-5782
Address: 1123 East Stadium Drive,
Logan, Utah, 84341

Summary

- Love teaching, programming and problem solving, passionate to learn new languages and tools.
- 2 years' experience as a .NET developer and Java Eclipse, developed a real world web application, and familiar with the Software Life Cycle Development and design patterns.
- Interested in Distributed System, Distributed Transaction Processing Systems, Cloud Computing and Web development, Information Retrieval.
- Like research that explores how a combination of software system architectures, automated software tools, and empirically derived programming guidelines can assist the programmer in developing and optimizing with the focus on Object-Oriented Programming, Aspect-Oriented Programming, Design Patterns, Software quality, information quality, and data analytics over large-scale distributed information systems like the Web, and Social Media.
- Like movies, nature, swimming, traveling and volunteering work.

Personal Data

Current Position: PhD Student and Graduate Teaching Assistant
Date of Birth: April 23rd, 1985
Citizenship: Jordanian
Marital Status: Married
Language Proficiency: Arabic and English
Email: anas.alsobeh@usu.edu
Github: https://github.com/aalsobeh
bitBucket: https://bitbucket.org/anas_cis
Linkedin: https://www.linkedin.com/home?trk=nav_responsive_tab_home

Education

- 2015 (expected) Ph.D., Computer Science, Utah State University (USU), USA. GPA: 3.94
- 2010 M.S. Computer Information System, Yarmouk University (YU), Irbid, Jordan. GPA: 88.2
- 2007 B. Sc., Computer Information System/ Yarmouk University, Irbid, Jordan. GPA: 86.2

Title of Ph.D. Dissertation

- Improving Reuse of Distributed Transaction Software with Transaction-Aware Aspects.

Certificates, Awards, and Memberships

- Member of USU Honored Graduate Students.
- First Rank of the 12 Master Computer Information Systems Student at Yarmouk University for the Academic Year 2007

- Ph.D. Scholarship from Yarmouk University (Jordan)
- Full Tuition Award, Utah State University.
- Member of SIGMOD’14 Committee

Funded Research Projects

- Ph.D. Research. A framework for weaving crosscutting concerns in distributed transactions, USU Graduate Studies and Multidimensional Software Creations LLC.
- A Distributed Health Data System, Early Childhood Collaboration Systems (ECCS). Multidimensional Software Creations, LLC.

Programming Languages and Skills

- C++, C#, and Visual Basic.NET (2003-2015). And profession in many powerful programming concepts and data structures.
- Java and AspectJ (Eclipse).
- Oracle (SQL, PL\SQL, FORMS).
- HTML, XHTML, PHP and Flash Mx.
- Databases (Microsoft SQL, Sybase, PostgreSQL, MySQL, Oracle).
- Web Search Engines (Design and Implementation)
- UML (Visual Paradigm, Rational Rose, ArgoUML, etc.).
- JBoss Application Server, Enterprise Application Platform (EAP)
- Atomikos and Bitronix Transaction Managers
- Amazon AWS (EC2, S3, and EMR), AWS SDK (Window Server), RDS (Rational Database, e.g., Mysql)
- Vitruvian: A framework for distributed application development

Experience

- Jan 2011 – Present, Graduate Research/Teaching Assistant, Computer Science Department, Utah State University, Logan, Utah, USA.
    - Responsibilities:
        - Teaching Assistant/ Develop curriculum in all areas: preparing lectures, assignments, and exams, holding office hours, grading exams, grading the assignments and assigning final grades for several classes:
            - CS 1030 - Foundation of Computer Science.
            - CS 1400 - C++ programming Language.
            - CS 1405 – C++ Lab.
            - CS 2610 - Web Development.
            - CS 3420 - Programming Language C# and .Net.
            - CS 5700 - Object Oriented and Software Design.
        - Teaching classes:
            - Object-oriented Software Design – UML and Design Patterns
        - Research Assistant/ Develop governmental projects and research in Distributed Systems, Integrated Application, Service-Oriented Programming (Vitruvian), Distributed Transaction, Design Pattern, Testing Distributed Applications, and Aspect-Oriented Programming (AOP).
- May, 2012 – December 2013, Software Engineer, Utah State University, USA
    - Online Course Evaluation System. The system provides help to the education department to evaluate their online courses and teaching more effectively. It allows them to collect and analyzing a wide variety of information from the feedback from professors and students.
        - Responsibilities: Team Manager

        - model and implement a relational database, design, implement, test webpages, create a range of reports and graphs, communicate with the users to understand their needs and adjust the system to meet their needs
    - Technical Skill Set: UML, Visual Paradigm, SQL, MsSQL, ASP.net, Microsoft Server 2008. HTML, CSS, JavaScript, jQuery.
- January 2010 – January 2011, Lecturer, Yarmouk University, Jordan
    - Teaching classes: Introduction of Information System, Computer Skill, and Oracle Labs.
- 2007- May 2010: Full-Time Teacher, Ministry of Education, Jordan
    - Teaching Computer Courses for Secondary and Primary grades.

Projects and Software Applications

- TransJ: A framework for weaving crosscutting concerns in distributed transactions. TransJ weaves crosscutting concerns for distributed transactions using Aspect-Oriented Programming (AOP) languages and allows the developer to implement transaction-related concerns in simple, well-known transaction constructs such as begin, commit, abort, lock, release, etc. which are hopefully more maintainable and reusable. TransJ was developed in AspectJ/Java.
- Early Childhood Collaboration Systems (ECCS). Sponsored by Multidimensional Software Creations (MDSC). I was worked on a policy enforcement services the ECCS that ensures confidentiality in the presentation of protected research data. The Early Childhood collaboration System (ECCS) is a distributed health data system which provides coordinated, de-identified healthcare information to various types of data consumers.
- Utah Preschool Outcomes Data System (UPOD). Sponsored by Multidimensional Software Creations (MDSC). The Baby and Toddler Online Tracking System (BTOTS) keeps track of the children receiving EI Part-C services. The data about children who are likely to need preschool special education services are sent from BTOTS to a system within the USOE.
- Transition from Early-intervention Data Input System (TEDI) - Distributed Systems and Web based Application. Sponsored by Multidimensional Software Creations (MDSC). This system, called Transition from Early-Intervention to Preschool Data Input System (TEDI), is responsible for tracking children already in the EI Part-C system who are entering EI Part-B up to the time the implementation of their IEP is started, e.g., they start receiving special education from EI Part-B. Once a child starts receiving special education from EI Part-B, the information is loaded into UPOD, and teachers start to track said child's progress. Figure 1 depicts the relationships among these three systems.
- Distributed Transaction Applications (Jboss Application Server (JbossAS), Java Application API (JTA), Enterprise Java beans (EJB), Java persistence application (JPA), and Servlet)
- I developed three distributed applications: Online Transaction Bank System, Conference Registration System, and Gadget Manufacturing System.
- Online-Ticket Distributed System- RMI, .Net Remoting, and Web Services (C#).
- Vitruvian: A Service Oriented Framework. Sponsored by Multidimensional Software Creations (MDSC). I worked with developing and refactoring a service oriented framework, in C#. This service-oriented architecture provides solutions to common problems encountered in software development. These solutions are packaged into reusable services that reduce development time for future projects. The framework provides services such as customized logging, serialization, communication and distribution. In general, the problems of this framework that are a distribution of using object replication and synchronization, using an object-oriented model to maintain the normal form of the data throughout the business and presentation layers.
- A Cloud-based Simulation of Disease Tracking System using AWS Amazon.
- A Distributed Simulation of Virtual Water Game (C#) - A distributed multiplayer game. A multi-player game where a mediator supervises several players on a shared game board. All of them act like

independent distributed components and interact among themselves using a standard set of communication protocols and well defined rules.

Research Interests

- Aspect Oriented Programming (AOP),
- Distributed Systems (DSs),
- Transaction Processing Systems (TPSs),
- Information Retrieval (IR),
- Software Quality and Metrics,
- Multilingual Queries Evaluation on Web Search Engines

Publications

- Anas M. R. AlSobeh, Stephen W. Clyde, Transaction-Aware Aspects with TransJ: Initial Experiment Show Promising Improvements in Reusability of Distributed Transactions, (selected to be published OOPSLA 2016).
- Anas M. R. AlSobeh, Stephen W. Clyde, TransJ: Independent Abstract Framework for Waving Aspect in Distributed Transaction Systems, (selected to be published *International Journal of Advanced Research in Computer Science and Software Engineering*, 2016).
- Anas M. R. AlSobeh, Stephen W. Clyde, Unified Conceptual Model for Joinpoints in Distributed Transactions, The Ninth International Conference on Software Engineering Advances, ICSEA'14. France, Nice, October 2014. ISBN: 978-1-61208-367-4.
- Anas AlSobeh, Stephen W Clyde, Independent abstract framework for Distributed Transaction, 11th Graduate Research Symposium, Poster Presentation, Research Graduate Studies Utah State University, 2015.
- Anas AlSobeh, Stephen W Clyde, Distributed Transaction Conceptual Model, 10th Graduate Research Symposium, Poster Presentation, Research Graduate Studies Utah State University, 2014.
- Al-Kabi, M.; Wahbeh, A.; Alsobeh, A.; Al-Eroud, A.; Alsmadi, I. Examining Web Search Trends across Arab Countries, The Arabian Journal for Science and Engineering B: Engineering, 2012.
- Anas AlSobh, Ahmed AlEroud, Muhammad AlKabi, and Izzat AlSmadi,A multilingual and location evaluation of search engines for websites and searched for keywords, Brazilian Journal of Information Science, 2011.

Course Work

- Distributed Systems Design (A)
- Aspect-Oriented Programming (A)
- Web Information Retrieval (A)
- Management Web-based Systems (A)
- Object-oriented Programming for Software Development (A)
- Software Engineering with Project (A)
- Software Testing and Research (A)
- Patterns in Software Systems (A)
- Integrated Systems (A)
- Graphical User Interfaces (A-)
- Software Architectures (A)
- Distributed Database Systems (A)
- Distributed Network Programming (A)
- Fundamental and Advance Databases Systems (A)

References

- Dr. Stephen Clyde, Associate Professor, Computer Science Department, Utah State University, Logan, UT 84321. Email: swc@mdsc.com or Stephen.Clyde@usu.edu.
- Dr. Curtis Dyreson, Associate Professor, Computer Science Department, Utah State University, Logan, Utah 84321. Email: Curtis.Dyreson@usu.edu.
- Dr. Izzat Alsmadi, Associate Professor, Computer Science Department, Boise State University, Boise, Idaho. Email: alsmadi@gmail.com.
- Dr. Mohammad Alkabi, Assistant Professor, Mathematical Department, Isra University, AlZarqa, Jordan. Email: mohammadAlkabi@yahoo.com.