Evaluating the Expressivity of AspectJ in Implementing a Reusable Framework for the ACID Properties of Transactional Objects

By

Ekwa J. Duala-Ekoko

School of Computer Science

McGill University, Montreal

June 2006

A THESIS SUBMITTED TO MCGILL UNIVERSITY IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

Copyright© 2006 Ekwa J. Duala-Ekoko

Abstract

Aspect-Oriented Programming (AOP) continues to gain increasing popularity in both academia and industry for its effectiveness in localizing and modularizing crosscutting concerns. Two recurring criticisms of AOP tools are their deficiency (or inflexibility) in language features and the potential performance overhead that may be incurred from adopting this technology. Using *AspectJ* as the target AOP tool, I investigate the language features necessary to support aspect dependencies, aspect interactions, aspect interference and aspect reuse in the context of implementing the ACID properties of transactional objects in a flexible and reusable way. Five encountered limitations are identified, namely; lack of support for inter-aspect configurability, lack of support for runtime disabling and re-enabling of pointcuts on a per-object basis, lack of support for per-instance association of aspects, lack of support for stronger and intuitive aspect-to-class binding of reusable static crosscutting behaviours, and the reflection/super-class method execution dilemma. Finally, I discuss the deficiencies of work-around solutions and suggest potential language improvements for addressing these limitations.

Résumé

De plus en plus de personnes du monde académique et industriel utilisent la programmation orienté-aspect (AOP) pour mieux structurer et modulariser leur code source. Beaucoup d'environments de programmation orientés-aspect ont été critiqués à cause de l'inflexibilié de leur language AOP et à cause du ralentissement de la vitesse d'exécution qui resulte de leur utilisation. Cette these évalue la capacité du language orienté-aspect AspectJ à exprimer les dépendences entre aspects, à exprimer et résoudre les interferences entre aspects, et à écrire du code orienté-aspect réutilisable. L'évaluation est effectuée en implémentant AspectOPTIMA, un cadre applicatifs qui guaranti les propriétés ACID (atomicité, cohérence, isolation et durabilité) pour les objets transactionnels. Cinq limitations du language AspectJ ont été identifiées: le language n'offre pas de méchanisme pour spécifier les dépendances entre aspects, le language ne permet pas d'associer un aspect à un objet (mais seulement à une classe), le language ne permet pas d'activer et de désactiver un aspect pendant l'execution du programme, les introductions statiques d'attributs dans une classe fait par un aspect ne se comportent pas comme les attributs standards face à l'héritage, et l'incompatibilité entre l'utilisation de la réflection et les aspects. La thèse explique également quelles concessions ont dû être faites pour implémenter AspectOPTIMA malgré ces limitations. En conclusion, la thèse suggère des améliorations possible pour le language AspectJ.

Acknowledgements

I am truly grateful to all those who have supported me directly or indirectly throughout my Master's program. The support and guidance I received from my thesis supervisor - Jörg Kienzle – during the thesis phase of this program was invaluable and greatly appreciated. Many thanks to my family and wife in particular for the countless sacrifices made during this phase of my life.

Dedication

To my dad - for his consistent support throughout my university life.

Table of Contents

List of Tables

List of Figures

Chapter 1 ~ Introduction

1.1 Motivation

Object-oriented programming [1] revolutionized the process of software development with its introduction of object abstraction, encapsulation, inheritance and polymorphism. These concepts have proven to be effective in modeling common hierarchical behaviours but fall short in modeling behaviours that spans across (i.e., crosscut) several unrelated modules. Attempts to implement such crosscutting concerns (i.e., system goals, concepts or areas of interest, non-functional requirements [14]) in object-oriented programming often results in systems that are difficult to reuse or maintain [2,3]. Aspect-oriented programming (AOP) [3,4] has been proposed as a new programming paradigm for addressing these deficiencies - resulting in a proliferation of AOP tools (*AspectJ* [4], *JbossAOP* [5], *Spring AOP* [6], *CaesarJ* [7]). The concepts and constructs of AOP have proven effective in localizing and modularizing crosscutting concerns; and consequently facilitating their reuse and evolution.

As expected of any new technology, the AOP user community continues to apply these concepts and tools in their respective domains and proposes new aspect-oriented language features to address their needs. However, the examples used for justifying new language features have been criticized as being too specialized to be convincing. The Software Engineering Lab at McGill University proposed a case study [8] to "*serve as a "benchmark" for evaluating new AOP approaches, programming language features and aspect-oriented software development and modeling approaches in general*". The case study was a by-product of an ongoing work to migrate OPTIMA [9] – a framework that provides transaction support for concurrent object-oriented programming languages - from an object-oriented to an aspect-oriented platform. The case study proposed a language-independent decomposition of the necessary runtime support to implement the ACID properties (*Atomicity*, *Consistency*, *Isolation* and *Durability*) of transactional objects used in transactions into reusable aspects. It then describes how these aspects can be recomposed to implement various *Concurrency Control* and *Recovery* strategies for transactional objects. The language independent design of this case study makes it ideal for evaluating AOP tools.

This thesis has two objectives. First, I verify the effectiveness of the design proposed in the case study; that is, does the proposed decomposition of *Concurrency Control* and *Recovery* into reusable aspects effectively capture all the required functionalities and can these aspects be seamlessly implemented in an AOP platform? Second, I evaluate the expressiveness of the language features of the current state of the art AOP tool – *AspectJ* - in the context of the case study. Simple stated, I attempt to answer the following question: Are the current programming language features of AspectJ (*version 1.5.0*) adequate for implementing a reusable framework for the ACID Properties of Transactional Objects (hereafter called the AspectOPTIMA framework)? If not, what are the encountered language limitations and how can these limitations be resolved? My definition of a reusable framework of aspects is comparable to the oblivious property suggested in [21] – i.e., developers do not have to modify an existing system to accommodate these aspects and the base application is completely ignorant of their existence. The enormous benefits of aspect-oriented software development (AOSD) in general and AOP in particular would be inconsequential to a specific domain unless the current AOP tools provide adequate and flexible programming language features to make this possible. The investigation of these questions will therefore take us a step closer to broadening the application base of both AOSD and AOP technologies.

1.2 Summary of Contributions

AspectJ has come a long way since its inception and its most recent release (AspectJ 5) is a testament of the commitment to broaden its application base. Notwithstanding, I identify several limitations in the context of the case study, namely:

- *Lack of support for runtime disabling and re-enabling of pointcuts*

  The *if(BooleanExpression)* pointcut of AspectJ is often promoted as the construct for achieving runtime disabling and re-enabling of aspects but this claim is only partially true. At best, this poincut supports the disabling and re-enabling of an advice within an aspect at runtime, not the join points of interest to which the advice is to be applied. This means that the join point(s) of the advice in question would still be intercepted but its execution is conditional on the value of *BooleanExpression.* This thesis argues that the performance overhead of unnecessarily intercepting join points and the evaluation of *if(BooleanExpression)* is not optimal for performance-sensitive applications. I propose new language constructs for facilitating runtime disabling and re-enabling of pointcuts (and consequently, target join points) so as to mitigate this deficiency.

- *Lack of support for per-instance association of aspects*

Aspects in AspectJ are statically deployed. That is, once compiled with a base class, the aspects become effective in every object of that class. Consequently, the application of different concurrency control and recovery strategy aspects to different *objects* of the *target application* is not an option. I highlight a need for the support of this functionality, suggest a potential solution and discuss some challenges of supporting per-instance association of aspects.

- *Lack of support for explicit inter-aspect configurability*

The functionality of some of the aspects in the case study requires the presence of other aspects; hence, such aspects should be able to express their need for other aspects while preserving obliviousness. AspectJ does not currently provide explicit support for inter-aspect configuration. I demonstrate how this can be achieved with the use of interfaces and discuss its limitations.

- *Weak aspect-to-class binding*

Abstract introduction [19,20] has been promoted as the strategy for creating reusable static crosscutting aspects that can be used in different contexts. This sounds like the perfect strategy for implementing the reusable concurrency control and recovery aspects proposed in the case study since these aspects will obviously be used in various contexts. I expose the pitfalls of this strategy and propose potential solutions.

- *Reflection/Super-class method execution dilemma*

  The method call pointcut (*call(MethodPattern)*) of AspectJ does not pick out reflective calls; a deliberate decision made by the AspectJ team not to "*delve into the Java reflection library to implement call semantics*" [9]. Developers are advised to use the method execution pointcut (*execution(MethodPattern)*) instead. The method execution pointcut on the other hand does not pick out the execution of non-overridden inherited methods when the context and target is the subclass. Developers must therefore choose between capturing reflective calls or the execution of non-overridden inherited methods, but both functionalities can not be provided simultaneously. I discuss the ramifications of this dilemma in the context of implementing a reusable aspect-oriented framework of transactional objects and suggest potential language improvements.

In addition to these AspectJ limitations, this work also makes the following contributions:

- It provides a tertiary contribution to the proposed case study by refining the decomposition and dependencies between the aspects.
- It validates the decomposition proposed in the case study.
- It provides a reusable aspect-oriented framework for the ACID properties of transactional objects - AspectOPTIMA. I discuss at a higher level how this was achieved and provide a stripped-down version of the implementation.

1.3 Thesis outline

The motivation and contributions of this thesis have been discussed in this chapter. Chapter 2 covers the fundamentals of transactions (transactional objects, ACID properties of transactions, concurrency control and recovery strategies) and AOP as it applies in the context of the case study and this thesis. A summary of the case study is presented in chapter 3. I introduce the implementation platform (Java^{TM} and AspectJ) in chapter 4. In chapter 5, I discuss the implementations of the aspects presented in the case study, critiquing each implementation and highlighting the encountered limitations. The encountered AspectJ limitations and potential improvements are discussed in chapter 6. Chapter 7 covers related work; chapter 8 contains the conclusions of the thesis and future work.

Chapter 2 ~ Fundamentals of Transactions and Aspect-Oriented Programming

This thesis brings together concepts from two different domains - transactions and aspect-oriented programming. This chapter introduces those concepts of these domains used in this thesis. Section 2.1 presents the fundamentals of transactions (transactional objects, ACID properties of transactions, concurrency control and recovery strategies). Aspect-oriented programming is introduced in section 2.2 of this chapter.

2.1 Transactions

2.1.1 Transactional Objects

A transaction [10,11] groups together operations involving one or more data objects (also known as *transactional objects*) that must either succeed or fail as a group. This ensures that the execution of these operations on transactional objects appears indivisible from the perspective of concurrent competing transactions. Three standard operations are used for marking transaction boundaries: *begin*, *commit* and *abort*.

The *begin* operation is used for signalling the beginning of a new transaction or sub-transaction. A transaction *abort* may be triggered voluntarily or involuntarily (in event of an exception) during the execution of a transaction. Upon *abort*, all the changes (*writes* or *updates*) made on the accessed transactional objects by the aborting transaction must be undone (also known as *rollback*). Upon a successful completion, all the modifications made by the committing transaction on transactional objects become permanent and visible to other transactions.

In classical transaction models, each transaction is executed by a single thread. More advanced models (such as *Open Multi-Threaded Transaction model* [11]) allow several threads to jointly participate in the execution of a transaction. The AspectOPTIMA framework provides support for both classical and advanced transaction models.

2.1.2 The ACID Properties of Transactions

Frameworks providing transaction support must be able to detect and resolve the execution of conflicting concurrent operations on transactional objects in order to preserve data consistency. To achieve these, frameworks must enforce the famous *ACID properties (Atomicity, Consistency, Isolation and Durability)* of transactions.

Atomicity

The *Atomicity* property guarantees the execution of either all or none of the operations within a transaction; hence, its synonymity to the *all-or-nothing* (or *at-most-once*) property of transactions. The net effect from an external viewpoint should be a jump from the initial state to the result state (in the event of a successful *transaction commit*) or no state change (in the event of a *transaction abort*). The execution of only a subset of the operations of a transaction is not acceptable as this may place the system in an inconsistent state. Consequently, *Atomicity* is said to be unconditional, i.e. it must hold under every potential catastrophic circumstances - including a crash of the operating system.

Consistency

Given a consistent state to start with, the *Consistency* property guarantees that the execution of a transaction (whether successful or not) will produce another consistent state. The results from a transaction must satisfy the validation constraints of the target application to be considered consistent. Inconsistent intermediate states within a transaction do not pose a problem since these are not visible to other transactions. This property is considered impossible to achieve without explicit programmer support, hence there is great reliance on the application developer to write consistency-preserving transactions.

Isolation

The *isolation* property prevents interference between concurrent executing transactions even when they access a common set of transactional objects. In other words, all the modifications made by a transaction on transactional objects cannot be based on data computed by other transactions still in progress. Consequently, the results produced by concurrently executing a set of transactions should be equivalent to the result produced by executing the same set of transactions in some arbitrary sequential order.

Durability

The *Durability* property guarantees that the results of successfully committed transactions survive program termination or system crash, even if the computer crashes immediately after a commit. That is, upon a successful transaction commit, the system must be able to re-establish its results (either by re-executing the same sequence of operations in event of an immediate system failure or by retrieving it from a stable storage) irrespective of subsequent failures.

2.1.3 Concurrency Control and Recovery

Transactions interact with transactional objects through well-defined public interfaces in properly designed systems. Implementing the ACID properties of transactional objects involves intercepting these interactions and performing the appropriate pre-actions and post-actions. These activities have been traditionally divided in to *concurrency control* and *recovery*. This section introduces the concepts of concurrency control and recovery necessary for this thesis.

Concurrency Control

The *concurrency control* component of a transaction framework guarantees the isolation and consistency properties of transactions. To achieve these, the concurrency control component must be able to distinguish between *observer* operations (i.e., read-only operations) and *modifier* operations (i.e., write and update operations), and must also be able to detect and resolve the execution of conflicting operations. There are two main techniques for conflict detection [11]: *strict concurrency control* and *semantic-based concurrency* control, but the latter is out of the scope of this work.

Strict concurrency control (*Table 2.1*) is used in distinguishing between *read*, *write* and *update* operations. *Read* operations do not modify the state of transactional objects, hence, they do not conflict with other read operations. A *Write* (i.e., a write-only operation) or an *Update* (a read followed by a write) operation on the other hand conflicts with other writes, updates or read operations because they modify the state of the transactional object on which they are executed.

	Read	***Write***	***Update***
Read	No	Yes	Yes
Write	Yes	Yes	Yes
Update	Yes	Yes	Yes

Table 2.1: Strict Concurrency Control conflict table

Concurrency control can be achieved either pessimistically [10] or optimistically [12], each having its advantages and disadvantages.

Pessimistic Concurrency Control

This technique requires a transaction to obtain permission from the concurrency control manager associated with a transactional object before executing an operation on it. The concurrency control manager first checks if the execution of this operation would conflict with other operations in progress. If so, the calling transaction is blocked or aborted. Otherwise, the transaction is given the permission to proceed, with an implicit guarantee of isolation.

Optimistic Concurrency Control

This technique allows the execution of conflicting operations on a transactional object but only persists the results of those transactions that do not violate system consistency. In order to accomplish this, the execution of a transaction is divided into three phases: a *read phase*, a *validation phase* and a *conditional write phase* (Figure 2.1). A transaction executes its write and update operations on a local copy of a transactional object during the *read phase*. The results of a transaction are only made global in the *write phase* if the *validation phase* succeeds.

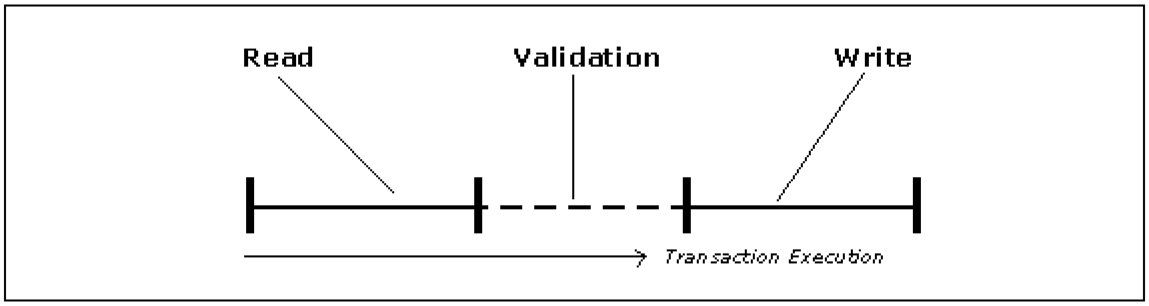


Figure 2.1: Three phases of a transaction

The *validation phase* can be further categorized as either *forward* or *backward* [13] based on the way in which conflicts are detected. *Forward validation* ensures that committing transactions do not invalidate the results of the transactions still in progress. *Backward validation* ensures that the result of a committing transaction has not been invalidated by recently committed transactions.

Recovery

The *recovery* component of a transaction framework guarantees the atomicity and durability of state modifications on transactional objects irrespective of system failures (*transaction abort* or *system crash*). In the event of a *transaction abort* (be it voluntarily or involuntarily), the recovery manager must undo all the modifications made by the aborting transaction. In the event of a *system crash*, the recovery manager must successfully abort all uncommitted transactions and undo all the modifications made on transactional objects by these transactions. It must then ensure that the results of transactions that committed before the crash are reflected in the appropriate transactional objects. Two types of techniques for performing updates and recovery of transactional objects have been identified in literature [11]: *in-place* and *deferred* update.

In-place update

This update strategy executes operations on behalf of the calling transactions on the *main copy* of a transactional object. This implies that all updates on transactional objects are instantly made global. The *undo* functionality (also known as *rollback*) is facilitated by taking a *snapshot* (also known as a *checkpoint*) of the state of a transactional object before it is modified. The states of transactional objects can be conveniently restored by rolling back to a previously established checkpoint - in the event of a transaction abort or system crash.

Deferred update

This strategy supports recovery/updates by creating a local copy of a transactional object per transaction the first time it executes a state modifying operation. Subsequent operations are executed on the local copy, making these changes invisible to the outside world. These changes are made global upon a successful transaction commit either by replacing the state of the main copy with that of the local copy or by re-executing these operations on the main copy. Undoing state modifications in the event of a transaction abort or system crash simply involves discarding the local copies of the appropriate transactions.

2.2 Aspect Oriented Programming (AOP)

Object-oriented programming (OOP) revolutionized the process of software development with its introduction of object abstraction, encapsulation, inheritance and polymorphism. These concepts have proven effective in modeling common hierarchical behaviours but fall short in modeling behaviours that spans across several unrelated modules. Attempts to implement such crosscutting behaviours in OOP often result in systems that are difficult to reuse or maintain. Aspect-oriented programming [3,4] has been proposed as a new programming paradigm for addressing these deficiencies.

AOP is not intended to be a replacement methodology to OOP but a complementary addition. It introduces new concepts and constructs that enable the modularization of crosscutting concerns, resulting in systems that are easier to understand, maintain and reuse. The most fundamental concept of AOP is the *Join Point Model* (JPM). The JPM specifies how crosscutting concerns interact with a base application. Specifically, it defines the locations in a base program were crosscutting concerns can be applied, a way for selecting these locations and a means of affecting the behaviour at these locations. The development of an AOP system typically involves three distinct phases [14]:

Aspectual Decomposition

This phase involves the identification of crosscutting and core system concerns (i.e., goals, concepts or areas of interest). Given a banking application for example, a developer may identify credit and debit activities as core concerns, and authentication, persistence and concurrency control as crosscutting concerns.

Concern Implementation

This phase involves the implementation of the concerns identified in phase one. The flexibility of AOP permits the independent/oblivious implementation of the core concerns in either a procedural (such as *C*) or an object-oriented (such as *C++* or *Java*) platform. The crosscutting concerns (i.e., aspects) are typically implemented in an AOP-extension of the base language.

Aspectual Re-composition

The rules for re-composing the concerns implemented in phase two into a final system are specified in this phase. These rules are typically specified in the same language in which the concerns were implemented and within an AOP class-like construct named *aspect*. Other AOP tools such as JBoss AOP and Spring AOP supports the specification of these rules in *XML* files. A *weaver* then re-composes these concerns using the specified weaving rules into a final system (see *figure 2.2*).

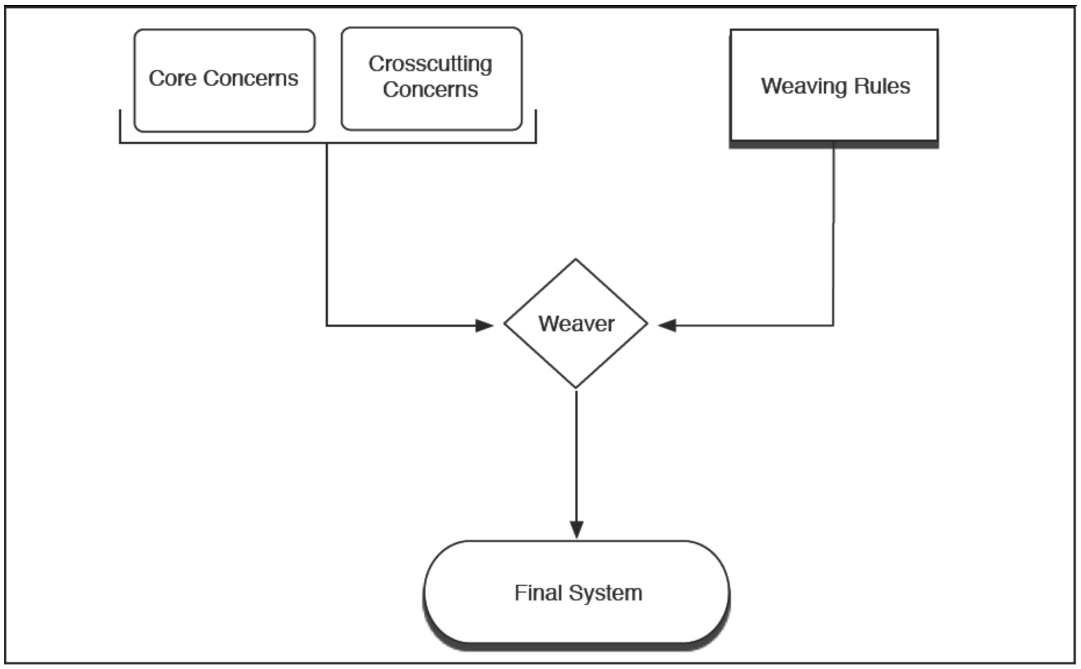


Figure 2.2: Re-composing concerns into final system

2.3 Summary

The foundational concepts of this thesis have been introduced in this chapter. Specifically, I introduced the ACID properties of transactions, concurrency control and recovery strategies, and the basics of AOP. In chapter 3, I show how the aspectual decomposition concept of AOP was employed by the Software Engineering Lab at McGill University to decompose the implementation of the ACID properties into reusable aspects.

Chapter 3 ~ The Case Study

This chapter presents the case study [8] proposed by the Software Engineering Lab at McGill University for evaluating AOP approaches and AOP language features. The case study argues that although concurrency control and recovery look like two separate concerns at a higher level, they cannot be completely separated at the implementation level. There exist both conflicts and common grounds between these two concerns. For instance, pessimistic concurrency control can only work with *in-place* update and both concerns must be able to distinguish *observer operations* from *modifier operations*.

Motivated by this incomplete separation of concerns, the study proposed a potential *aspectual decomposition* of concurrency control and recovery (presented in *section 3.1*) into well-defined reusable aspects. It then showed how these aspects can be re-composed to provide various concurrency control and recovery strategies (presented in *section 3.2*). This thesis provides a tertiary contribution to the case study - some of these aspects have been refined where necessary to achieve a more elegant and functional decomposition. The version of the case study discussed below reflects these contributions.

3.1 Aspectual Decomposition of Concurrency Control and Recovery

This section presents a brief description of each of the aspects proposed in the case study. Specifically, I discuss the motivation, dependencies (i.e., other aspects that the current aspect depends on or other aspects that require the functionality provided by the current aspect) and interferences (i.e., aspects that have to modify their behaviours in the presence of other aspects) of each aspect. The justifications of the reusability of the aspects can be found in [8] and the implementation details are separately discussed in *chapter 5* of this thesis.

3.1.1 AccessClassified

Motivation

The AspectOPTIMA framework must be able to identify the operations of a transactional object, which if executed concurrently may compromise the object's state consistency. To this end, the operations of transactional objects must be classified into three categories: *read operations* (i.e., operations that do not modifier the state of an object), *write operations* (i.e., write-only) and *update operations* (i.e., a read, followed by a write). The *AccessClassified* aspect provides this functionality.

Dependencies

- **Depends on**: None
- **Interferes with**: None
- **Is used by**: *Shared*, *Tracked*, *AutoRecoverable* and Concurrency Control

3.1.2 Named

Motivation

One of the fundamental properties of an object is its identity because it helps in distinguishing it from other objects. A memory reference is typically used to uniquely identify an object at runtime. Transactional objects by nature have need of a lifespan that is not tied either to the lifetime of a memory location or an application. Consequently, there must be a way for uniquely identifying transactional objects that will transcend program termination and the lifetime of an application.

The *Named* aspect provides this functionality. A transactional object should be given a name at creation time and the name should remain valid throughout its lifetime. It should be possible to obtain the name of a given object and to retrieve an object given its name.

Dependencies

- **Depends on**: None
- **Interferes with**: None
- **Is used by**: *Tracked*, *Persistent, Serializeable and Versioned*

3.1.3 Copyable

Motivation

An object encapsulates state. At times (e.g., as in recovery strategies using *deferred update*), it may be necessary to duplicate an object's state or to replace the state of one object with that of another object of the same class. These functionalities are provided by *Copyable*. This aspect should detect the presence of *Shared* and *Named*. The name of an object must not be changed by the state replacement operation and the state replacement or duplication of a *Shared* object should occur in mutual exclusion.

Dependencies

- **Depends on**: None
- **Interferes with**: *Shared* and *Named*
- **Is used by**: *Serializeable* and *Versioned*

3.1.4 Shared

Motivation

Transactional objects are shared data structures. Threads running concurrently within the same transaction may simultaneously execute conflicting operations on a transactional object - producing an inconsistent state. It is therefore necessary to prevent the threads that jointly participate in the execution of a transaction from concurrently modifying an object's state. The *Shared* aspect provides this functionality. It provides exclusive access of a transactional object to either a single state modification operation (*modifier*) or multiple concurrent read operations (*observers*) - assuming no modification operation is in progress. *Shared* depends on *AccessClassified* in order to distinguish *observer operations* from *modifier operations*.

Dependencies

- **Depends on**: *AccessClassified*
- **Interferes with**: None
- **Is used by**: Concurrency Control

3.1.5 Serializeable

Motivation

The state of a transactional object is not restricted to main memory. Certain functionalities (e.g., *Persistence*) require an object's state be moved to a different location such as a file or a database. The main memory representation of the object must therefore be transformed to

the appropriate representation of the destination location. The *Serializeable* aspect provides this functionality. It enables a transactional object to be able to write its state to a backend requiring varying representation formats, read its state from a back end and create a new object - initialized with the state read from a backend.
Serializeable relies on *Copyable* in replacing the state of an object with that of a previously serialized object of the same class. *Serializeable* also interferes with *Shared* and *Named*. A shared transactional object should only be serialized when there is no other transaction modifying it. This aspect should also detect the presence of *Named* and serialize an object's name together with its state.

Dependencies

- **Depends on**: *Copyable*
- **Interferes with**: *Shared* and *Named*
- **Is used by**: *Persistent*

3.1.6 Versioned

Motivation

State modifications made by a transaction on transactional objects must be isolated from other transactions until the outcome of the transaction is known. To facilitate this, each transaction must have its own *view* of the transactional objects it accesses and threads should only see the updates made by other participants of the same transaction but not the updates made from within other transactions. Consequently, multi-version concurrency control strategies, as well as snapshot-based recovery techniques have to create multiple copies of the state of a transactional object. The *Versioned* aspect provides this functionality.

A Versioned object encapsulates several versions of the state of a transactional object, with one version designated as the *main version*. Versions are linked to the *views* of transactions. A method invocation on a transactional object is either directed to the version of a particular view - if the invoking transaction has a view - or to the *main version* - otherwise. *Versioned* relies on *Copyable* (for duplicating the state of an object) and *Named* (for uniquely identifying an object). It indirectly interferes with *Shared* (a new version should only be created when an object's state is not being modified by another transaction) but *Copyable* has already taken care of this.

Dependencies

- **Depends on**: *Copyable* and *Named*
- **Interferes with**: None
- **Is used by**: *Recoverable* and Concurrency Control

3.1.7 Tracked

Motivation

The AspectOPTIMA framework must keep track of all the transactional objects accessed by transactions (or threads) in order to effectively ensure the ACID properties of transactions. For instance, the list of modified transactional objects is required to support *rollbacks* or *global-updates* in the event of a transaction abort or a transaction commit respectively. The *Tracked* aspect provides this functionality. It enables threads to define regions in which object accesses can be monitored in a generic way. The tracked region is delimited by begin and end operations. A thread should be able to obtain all *read*, *written-to* or *updated* transactional objects for the given region at any point in time. *Tracked* relies on *AccessClassified* (to distinguish *observer* operations from *modifier* operations) and *Named* (to avoid duplicates). *Tracked* should detect the presence of *Versioned* to avoid tracking different versions of the same object.

Dependencies

- **Depends on**: *AccessClassified* and *Named*
- **Interferes with**: *Versioned*
- **Is used by**: Concurrency Control and Recovery

3.1.8 Recoverable

Motivation

The AspectOPTIMA framework must be able to undo (also known as *rollback*) all state modifications made on transactional objects by aborting transactions in order to ensure the *all-or-nothing* property of transactions. The *Recoverable* aspect provides this functionality. It provides transactional objects with the ability to save their state (also known as establishing a *checkpoint*) and restore it at a later time, if need be. It should be possible to establish multiple *checkpoints* and to rollback an object's state to any of the previously established *checkpoints*. The *Recoverable* aspect should also support both *in-place* and *deferred* updates.

Recoverable depends on *Versioned* in establishing a checkpoint. It indirectly interferes with *Shared* (a *checkpoint* should only be established when an object's state is not being modified by another transaction) but *Versioned* has already taken care of this.

Dependencies

- **Depends on**: *Versioned*
- **Interferes with**: None
- **Is used by**: *AutoRecoverable* and Recovery.

3.1.9 AutoRecoverable

Motivation

A potential performance issue when establishing multiple checkpoints of an object is the possibility of no state changes between successive checkpoints. An optimal solution may be to establish a checkpoint only when it is determined that the execution of an operation would modify the object's state. The *AutoRecoverable* aspect provides this functionality. It allows a thread to define a region within which a checkpoint is automatically established before the execution of a state modification operation. This region is delimited by begin and end operations and all state modifications made within this region can be undone at any time. *AutoRecoverable* relies on *AccessClassified* (to distinguish *observer* operations from *modifier* operations) and *Recoverable* (to provide undo functionality).

Dependencies

- **Depends on**: *Recoverable* and *AccessClassified*
- **Interferes with**: None
- **Is used by**: Recovery.

3.1.10 Persistent

Motivation

The state of persistent objects must outlive program termination. To support this, persistent objects must be able to write their state to stable storage (such as a database [15] or a file) and subsequently restore the object's state based on the content of the storage device. A storage device should be specified at object creation time and it should be possible to destroy the object when no longer needed.

The *Persistent* aspect depends on *Serializeable* (for transforming the object's state into the appropriate format), *Copyable* (for restoring the state of an object) and *Named* (for designating a valid storage location on the storage device). It interferes with *Versioned* (i.e., of all the versions, only the main version should be persisted) and *Recoverable* (a recoverable object should be persisted together with its checkpoints). The indirect interference between *Persistent* and *Shared* (i.e., an object's state should only be persisted when it is not being modified by another transaction) has already been taken care of by *Serializeable*.

Dependencies

- **Depends on**: *Serializeable, Copyable* and *Named*
- **Interferes with**: *Versioned* and *Recoverable*
- **Is used by**: Recovery

3.1.11 Summary

I have discussed the decomposition of concurrency control and recovery into reusable aspects in this section. Specifically, I presented the motivation, dependencies and interferences of each aspect. Figure 3.1 shows a UML diagram of the dependencies and interferences between these aspects. The solid arrows depict dependencies and the broken arrows depict interferences. The *Concurrency Control* and *Recovery* aspects have been added to represent general concurrency control and recovery strategies. Those highlighted in grey represent implementation overlap between *Concurrency Control* and *Recovery*. Finally, aspects that have to intercept calls/executions to transactional objects are stereotyped <<*interceptor*>>.

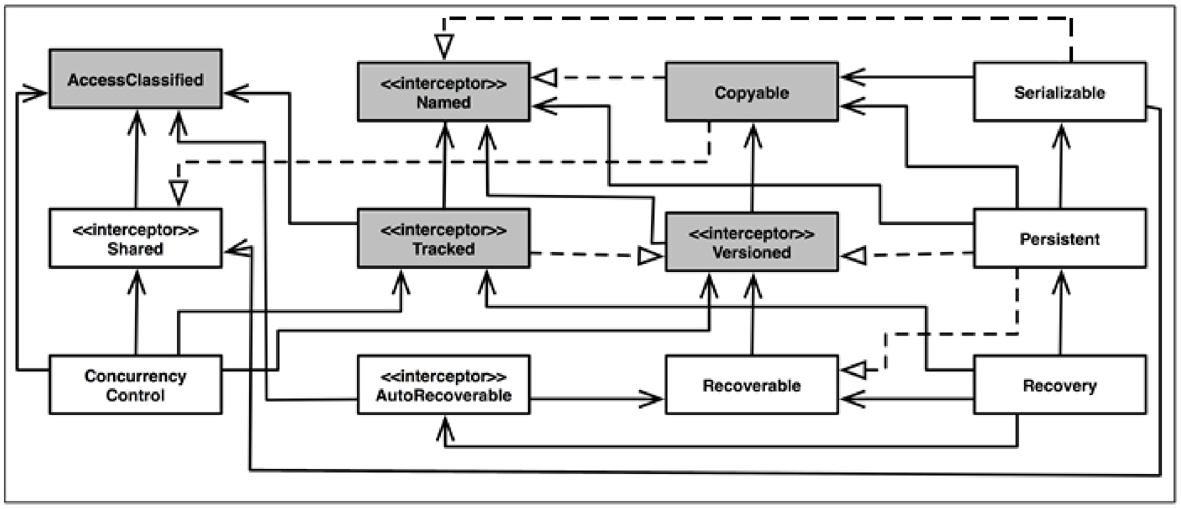


Figure 3.1: Aspect dependencies and interference

3.2 Aspectual Re-composition of Concurrency Control and Recovery

This section describes how the aforementioned aspects can be re-composed to achieve different concurrency control and recovery strategies for transactional objects. It is assumed that the transaction framework creates a *tracked zone*, a *recovery zone* and a *view* for each transaction when it begins, and that it ends these zones upon transaction abort or commit.

3.2.1 Pessimistic Lock-Based Concurrency Control with In-Place Update

The objective of the *LockBased* aspect is to provide a pessimistic lock-based concurrency control for transactional objects. When a transaction invokes an operation on a transactional object, this aspect obligates the transaction to obtain permission (i.e., a lock) from the AspectOPTIMA framework before execution the operation. Each public operation is assumed to have a *read*, *write* or an *update* lock associated with it. *LockBased* depends on *AccessClassified* in determining the type of lock requested by a given operation. The lock is only granted if it does not conflict with a lock held by another transaction currently in progress. Otherwise, the calling transaction is blocked till the requested lock is released. *LockBased* then selects an *in-place* update strategy by calling *Recoverable* before allowing the call to proceed.

The AspectOPTIMA framework needs to know all the transactional objects accessed by a transaction in order to be able to release the acquired transactional locks upon a transaction abort or commit. For this, *LockBased* depends on *Tracked*. *LockBased* also depends on *AutoRecoverable* (to prepare for *rollback* by establishing a checkpoint before state modification operations), on *Versioned* (to direct the operation call to the *main version* of the transactional object) and on *Shared* (to ensure that no two threads within a transaction can modify the object's state concurrently).

The *Shared* aspect releases the mutual exclusion lock immediately after the execution of the operation. Transactional locks, however, are held till the outcome of the transaction is known. The execution order of conflicting transactions is determined by the order in which transactional locks are granted. This implies that transactions acquire locks during their execution phase (*phase one*) and release them once the outcome of the transaction is known (*phase two*) – a process known as *two-phase* [13] locking. The sequence diagram below (Figure 3.2) shows how a call to a transactional object (TAObjects) is intercepted and how all the aspects involved in this concurrency control strategy collaborate to achieve the desired functionality.

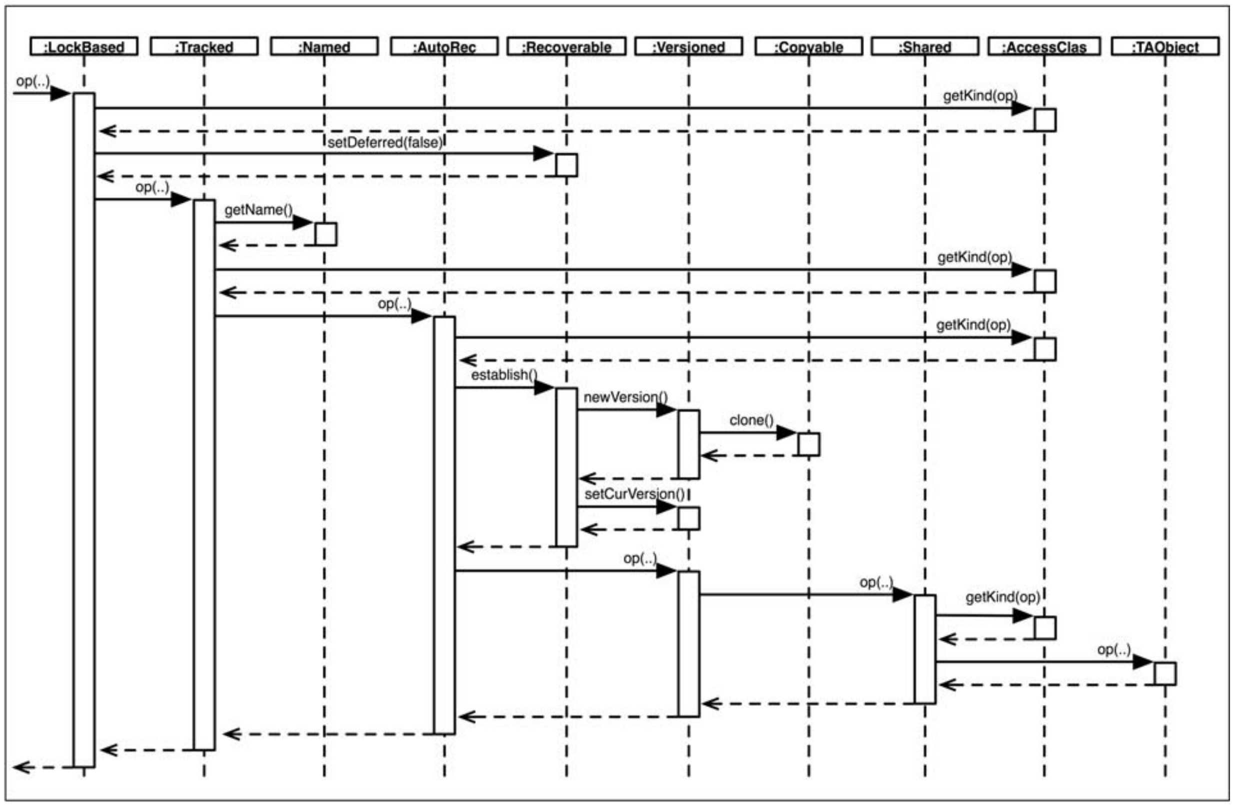


Figure 3.2: Aspect Collaboration for the *LockBased* Aspect

3.2.2 Pessimistic Multi-Version Lock-Based Concurrency Control with In-Place Update

A major weakness of the lock-based concurrency control strategy presented above (section 3.2.1) is that *read-only* transactions (*observers*) can be blocked by *modifier* transactions. This may greatly impact the performance of applications with many short-lived *observer* transactions and few long-lived *modifier* transactions. The *MultiVersion* aspect addresses this problem. The *MultiVersion* aspect encapsulates two sets of states per transactional object, namely: a *history of committed states* (HCS) and the *main version*. *Observer* operations are executed on the appropriate objects in the HCS whereas *modifier* operations are executed on the main version. The HCS is populated by creating new versions of a transactional object upon a successful transaction commit. Each object in the HCS is assigned a logical timestamp during which it state was valid.

Read-only transactions are assigned logical timestamps at creation time and they no longer have to acquire locks before observing the states of transactional objects. Notwithstanding, *MultiVersion* has to intercept *read-only* transactions, find a committed version with the highest timestamp that is lower than the transaction timestamp and assigns this version to the *view* of the invoking transaction. *MultiVersion* depends on *Versioned* to direct the call to the selected version and on *Tracked* - to record read accesses. The *AutoRecoverable* and *Shared* aspects are not needed for read-only transactions since these transactions do not modify the state of transactional objects.

Figure 3.3 illustrates the control flow when an operation is invoked on a *MultiVersion* transactional object. The versions *old1* to *old4* represents the history of committed states of the transactional object (TAObject). Ideally, the *Shared* aspect should be disabled for the versions *old1* to *old4* of the TAObject but enabled for the *main version* (*main*) to optimize performance.

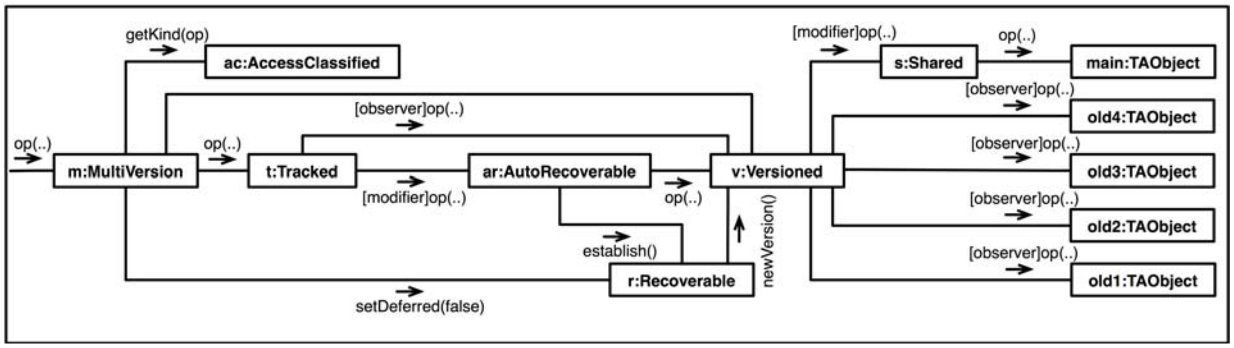


Figure 3.3: Control Flow for Multi-Version Concurrency Control

Modifier transactions are handled in the same way as in the *LockBased* aspect. First, *MultiVersion* attempts to acquire the appropriate lock (*Write* or *Update* lock) on the *main version* of the transactional object (TAObject). If granted, the transaction is allowed to proceed; otherwise, it blocks. *MultiVersion* then depends on *Tracked* to record object accesses, on *Recoverable* for selecting an *in-place* update strategy and on *AutoRecoverable* for establishing a checkpoint. *Versioned* then directs this call to the *main version* and *Shared* ensures mutual exclusion for concurrently executing conflicting threads. As before, *Shared* releases the lock immediately after the execution of the operation but the transactional locks are held till the outcome of the transaction is known.

3.2.3 Optimistic Concurrency Control with Deferred Update and Backward Validation

The *Optimistic* aspect implements an optimistic concurrency control strategy with deferred update and backward validation. As discussed in *section 2.1.3*, the execution of a transaction is divided up into three phases (*Read*, *Validation* and *Write*) when using optimistic concurrency control.

Read Phase

Transactions always read the most recently committed state of a transactional object during this phase. The *Optimistic* aspect intercepts and classifies method calls on transactional objects with the assistance of *AccessClassified*.

If the method call is an *observer* operation, it is first forwarded to *Tracked* to record object accesses. Then, *Versioned* forwards the call to the *main version* since it contains the most recently committed state.

The *Shared* and *AutoRecoverable* aspects are not necessary in this case because it is a *read-only* call.

If the method call is a *modifier* operation, *Optimistic* first selects a *deferred-update* strategy by calling *Recoverable*. The call is then forwarded to *Tracked* to record the object access. Next, *AutoRecoverable* creates a new version of the transactional object using the selected update strategy. Finally, *Versioned* forwards the call to the newly created version and *Shared* ensures mutual exclusion as usual. All subsequent reads performed by this transaction are executed on the local version of the transactional object.

AutoRecoverable creates a local version per-transaction for each concurrent *modification* call. Consequently, there might exist several uncommitted versions of a transaction object at a given time, each belonging to a different transaction. Figure 3.4 illustrates the control flow through the aspects for *observer* and *modification* operations. We have four active transactions, each with its own local copy of the transaction object. *Observer* operations are executed on the *main version* and have no need for *AutoRecoverable* or *Shared*.

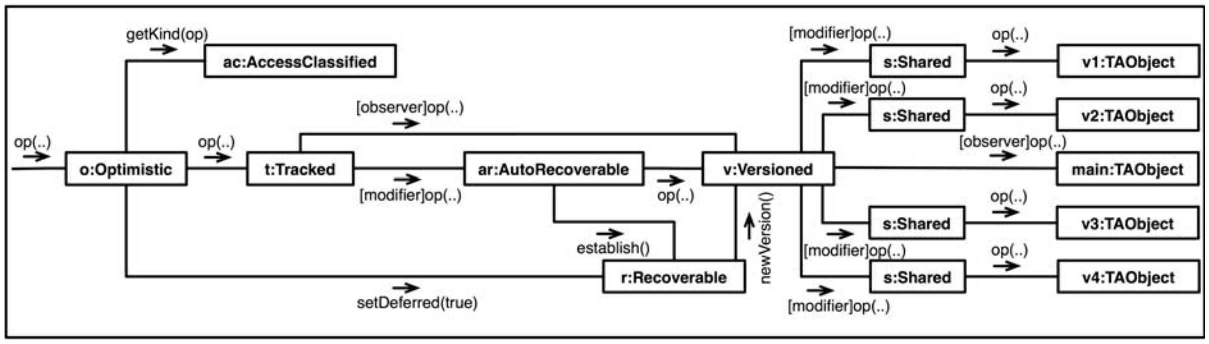


Figure 3.4: Control Flow for Optimistic Concurrency Control

Validation Phase

The results of transactions have to be successfully validated before they are committed. To achieve this, the *Optimistic* aspect requires the timestamp of the most recently committed transaction both when a transaction begins (*Tbegin*) and when a transaction is about to begin its validation phase (*Tend*). It then computes the union of all the transactional objects modified by transactions (excluding the current validating transaction) between *Tbegin* and *Tend* using *Tracked* and intersects this union with the set of transactional objects modified by the validating transaction.

A transaction successfully validates only if the intersection set is empty. It then receives a commit timestamp before proceeding to the write phase. Otherwise, validation is considered unsuccessful and the validating transaction is aborted. *Recoverabl*e is then informed to rollback the changes, resulting in the deletion of the local versions of the transaction.

Write Phase

This phase is responsible for making the results of a successfully validated transaction global and for discarding the local versions of a transactional object.

3.3 Summary

I have discussed the decomposition of the ACID properties of transactions into reusable aspects proposed by [8]. I also discussed how these aspects could be re-composed to achieve different concurrency control and recovery strategies for transactional objects. A surface examination of the decomposition and re-composition scenarios highlights some preliminary requirements that a typical AOP tool should provide:

- *Inter-aspect configurability(i.e., aspect dependencies)*

  Some aspects require the functionality of other aspects in order to function (e.g., *AutoRecoverable* depends on *Recoverable* and *AcessClassified*). An AOP tool should provide a construct for explicitly expressing these dependencies.

- *Inter-aspect ordering*

  The ability to define the execution order of aspects is crucial in certain cases. *Optimistic* for instance, requires *AutoRecoverable* to create a new version of a transactional before *Versioned* directs and executes the call on the appropriate version.

- *Per-instance aspect association*

  An often-desired functionality is the possibility to apply different concurrency control and recovery strategies to different objects of the same class. It should therefore be possible to apply *Lockbased*, *MultiVersion* and *Optimistic* to objects, instead of classes.

- *Dynamic aspects*

  There is no need of the *Shared* aspect on transactional objects that are accessed only by *observer* operations. The presence of *Shared* on such objects may be a cause for concern for performance sensitive applications. An AOP tool should therefore provide support for runtime enable or disabling of aspects.

Chapter 4 ~ Implementation Platform

This chapter introduces the concepts and constructs of *Java* and *AspectJ* that are of importance to this thesis. I introduce user-defined annotations (supported as of Java 5) in Section 4.1. These annotations are used for classifying the operations of transactional objects as *Read*, *Write* or *Update*. AspectJ is introduced in section 4.2.

4.1 Annotations

Before the advent of user-defined annotations, frameworks (such as Junit [16]) relied on naming conventions for identifying methods that require special treatment at runtime. This approach is restrictive and prone to implementation and evolutionary errors. As of release 5.0 (also known as Tiger) [17], the Java platform now provides a versatile approach for annotating program elements (fields, methods, parameters, constructors, local variables, packages, annotations, classes, interfaces and enumerations).

The declaration of an annotation takes an at-sign (@), followed by the *interface* keyword and the name of the annotation. Annotations also have a target program element, a retention policy (*Source*, *Class* or *Runtime*) and an inheritance policy. These policies are specified using *meta-annotations* (predefined annotations used for annotating other annotations). Java currently supports three types of annotations: *marker annotations* (i.e., annotations with no elements), *single-value* *annotations* (i.e., annotations with a single element) and *multi-value annotations* (i.e., annotations with multiple elements).

I use *marker annotations* in distinguishing between the *read*, *write* and *update* operations of a transactional object. Figure 4.1 shows the *Read* marker annotation used in this work. It is equivalent to the *Write* and *Update* annotations, except for the difference in name. As shown below, these annotations have a runtime retention policy (i.e., they are retained by the virtual machine so that they can be read reflectively at run-time), can be inherited (i.e., annotations on super-classes are automatically inherited by subclasses) and their target is method declarations.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Inherited
public @interface Read {}
```

Figure 4.1: Sample marker annotation

Unfortunately, method annotations on super-interfaces cannot be inherited by implementing classes [18]. The framework must therefore employ a different strategy for classifying methods inherited from super-interfaces to avoid the performance overhead that come from using worst-case classification. These strategies are discussed in the implementation of *AccessClassified* (section 5.3.1).

4.2 AspectJ

AspectJ [4] is an aspect-oriented extension of the Java programming language. It emerged from a research work at Xerox PARC aimed at modularizing crosscutting concerns and is currently considered to be the most mature AOP implementation. This section introduces those concepts and constructs of AspectJ that are used in this thesis (see [14] for an in-depth coverage on AspectJ). AspectJ supports two types of crosscutting behaviours: *dynamic* and *static* crosscutting. These crosscutting behaviours are encapsulated in an AspectJ class-like construct known as an *aspect*. Similar to a Java class, an *aspect* can contain both data members and method declaration but it cannot be explicitly instantiated.

4.2.1 Dynamic Crosscutting

Dynamic crosscutting techniques are used for defining behaviours that modify the runtime execution of a system either by augmenting or replacing it. I introduce the AspectJ constructs used for modifying a system's dynamic behaviour in this section.

Join point

Join points are well-defined points in the execution of a program. The integration of crosscutting concerns with base applications occurs at these points. A program's execution contains several join points but AspectJ exposes only the following: method call and execution, constructor call and execution, read or write access to a field, object and class initialization execution, exception handler execution, and advice execution. These join points also have states associated with them such as the current executing object, the target object or the list of arguments.

This thesis makes use of method call and execution, constructor call and execution, and advice execution join points. My discussion of join points is therefore limited to these.

Method and Constructor call join points

The method and constructor call join points are equivalent in definition. Both occur at the places where they are being invoked (*Figure 4.2*). The constructor call join point of the *Account* object occurs at the statement that requests the creation of a new object. Similarly, the method call join point of the *getBalance()* method of the *Account* object occurs at the point where it is being invoked.

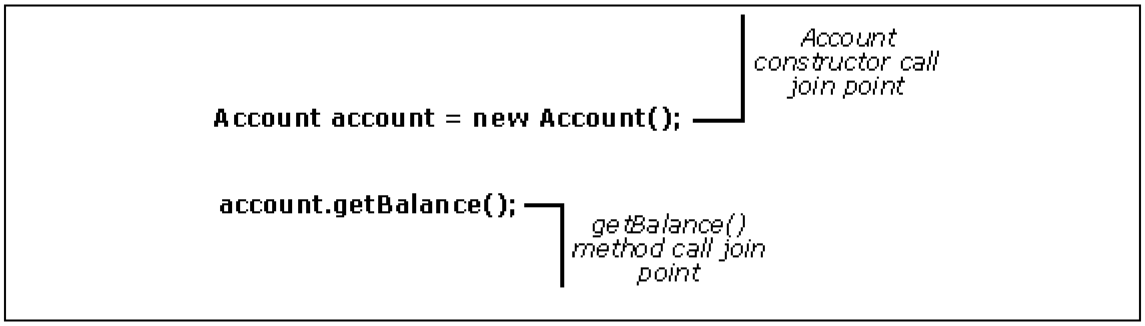
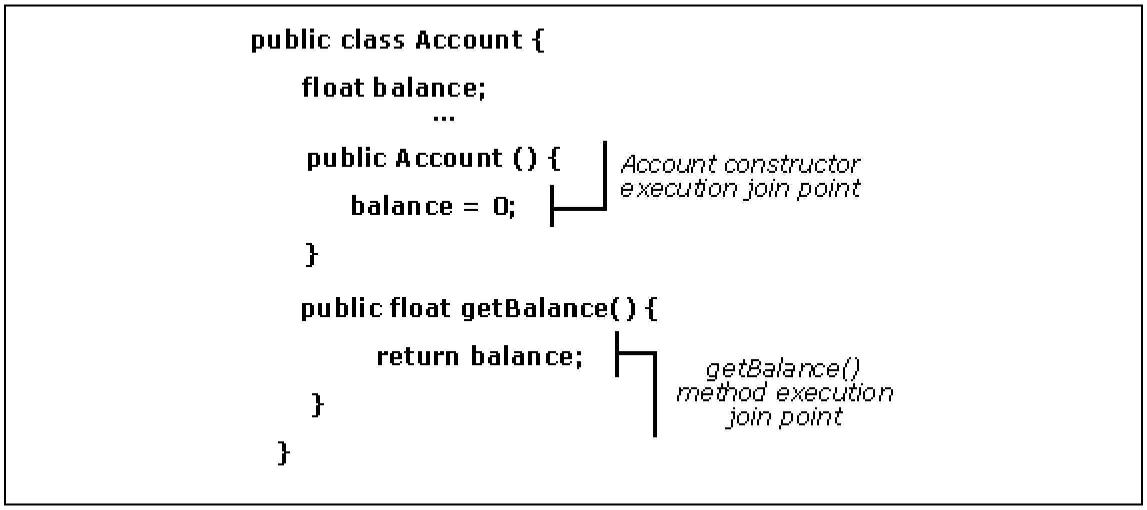

```
Account account = new Account();
account.getBalance();
```
Account constructor call join point
getBalance() method call join point

Figure 4.2: Method and Constructor call join points

Method, Constructor and Advice execution join points

The method, constructor and advice execution join points are also equivalent in definition. These occur when the code within the body of the corresponding construct executes. Figure 4.3 shows an example of the *getBalance()* method and the *Account* constructor execution join points of the Account object.



```
public class Account {
    float balance;
        ...
    public Account () {
        balance = 0;
    }
    public float getBalance( ) {
        return balance;
    }
}
```

Account constructor execution join point
getBalance() method execution join point

Figure 4.3: Method and Constructor execution join points

AspectJ provides a special reference variable – **thisJoinPoint** – that contains the dynamic information associated with an advised join point. This variable provides direct access to information such as the target object (**thisJoinPoint**.getTarget()), the current executing object (**thisJoinPoint**.getThis()), and method arguments (**thisJoinPoint**.getArgs()). Other information such as the name of the current executing method can also be extracted (indirectly) from this variable using the Java reflection API.

Pointcut

A pointcut is a construct used for capturing join points of interest and their associated context - such as the current executing object (*this(ObjectIdentifier)*), the target object of a call or execution (*target(ObjectIdentifier)*) and the arguments of the join point (*args(...)*). AspectJ supports both *named* and *anonymous* pointcuts. *Named* pointcuts are declared using the keyword *pointcut* and can be reused in multiple places. Pointcuts can also be composed using Boolean operators (AND, OR, NOT) to build other pointcuts. Below is the syntax of a named pointcut:

[*access modifier*] pointcut *pointcut-name* ([*arguments*]): *pointcut-definition*

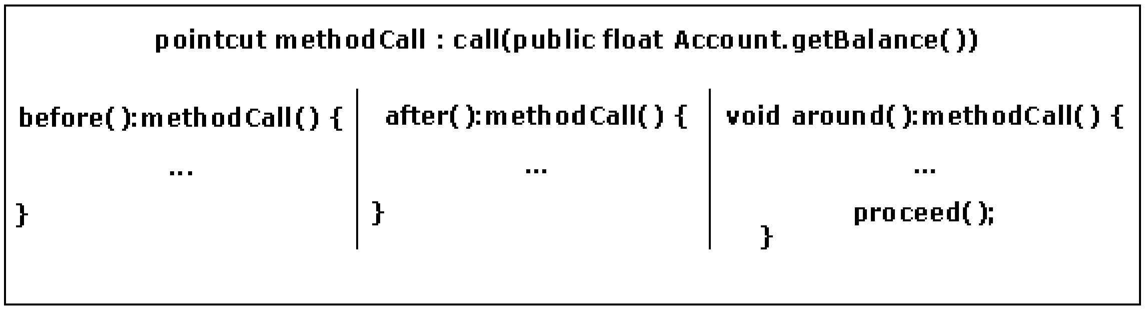
The *access modifier* can be public, private or protected; the *pointcut-name* can be any valid user-defined non-keyword and *arguments* are used for exposing context of interest to the advice. Table 4.1 introduces the *pointcut-definition* of interest to my work.

Pointcut-definition	**Description**
call(public * Account+.*(..))	Picks out all public method calls to Account or its subclasses, taking zero or more arguments.
execution(public * Account+.*(..))	Picks out all public method execution on Account or its subclasses, taking zero or more arguments.
call(public Account+.new(..))	Picks out public constructor calls to Account or its subclasses, taking zero or more arguments.
execution(public Account+.new(..))	Picks out public constructor execution on Account or its subclasses, taking zero or more arguments.
target(Account)	Picks out all join points where the target is an instance of Account.
Adviceexecution()	Picks out the execution join point of an advice.
!cflow(Adviceexecution())	Picks out all join points that are not in the control flow of the executing advice.
if(*BooleanExpression*)	Picks out all join point where the Boolean expression evaluates to true.

Table 4.1: Sample pointcut definitions

Advice

An advice defines the actions to be taken at the join point(s) captured by a pointcut. AspectJ supports three types of advice: *before*, *after* and *around* advice. The *before* advice runs just before the captured join point; the *after* advice runs immediately after the captured join point; the *around* advice surrounds the captured join point and has the ability to augment, bypass or allow its execution. The *after* advice comes in three flavours: *after returning* (i.e., after the successful execution of a join point), *after throwing* (i.e., after the advised join point throws an exception), and *after* (i.e., irrespective of the return status of the join point).



```
pointcut methodCall : call(public float Account.getBalance())

before():methodCall() {     after():methodCall() {     void around():methodCall() {
        ...                         ...                          ...
}                           }                                proceed();
                                                           }
```

Figure 4.4: Sample *before*, *after* and *around* advice

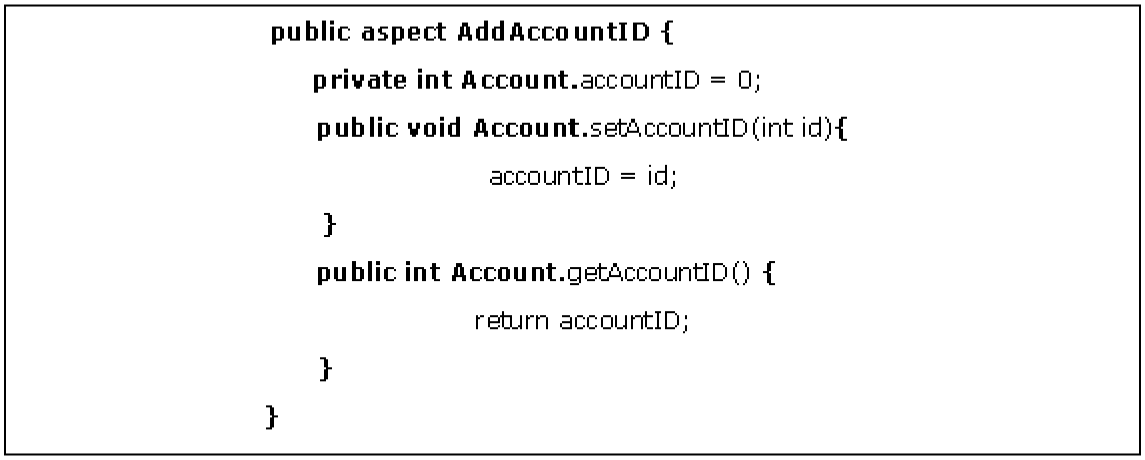
Figure 4.4 shows an example *before*, *after* and *around* advice for the pointcut *call(public float Account.getBalance())*. The *proceed()* statement of the *around* advice passes control back to the captured join point. Omitting this statement will bypass the captured join point.

4.2.2 Static Crosscutting

The static crosscutting constructs are used for modifying the static structure (i.e., classes, interfaces and aspects) of a system. I introduce the AspectJ constructs that support this type of crosscutting in this section.

Inter-type member declarations (also known as Introductions)

The *member introduction* concept of AspectJ is a feature that is extensively used in this thesis. This concept facilitates the introduction of *data members* and *methods* - with implementation - into classes and interfaces. Figure 4.5 illustrates the introduction of a private *accountID* field, with public setter (*public void setAccountID(int id)*) and getter (*public int getAccountID()*) methods into the *Account* class. This type of introduction is called *direct introduction* because the data members are introduced directly in the target class. *Indirect introduction* introduces data members into an interface and later binds the interface to the target class. A significant difference between *direct* and *indirect introduction* of relevance to this thesis is the association of the joint points of the introduced methods. The call and execution joint points of directly introduced methods are associated with the target classes whereas those of indirectly introduced methods are associated with the interface.

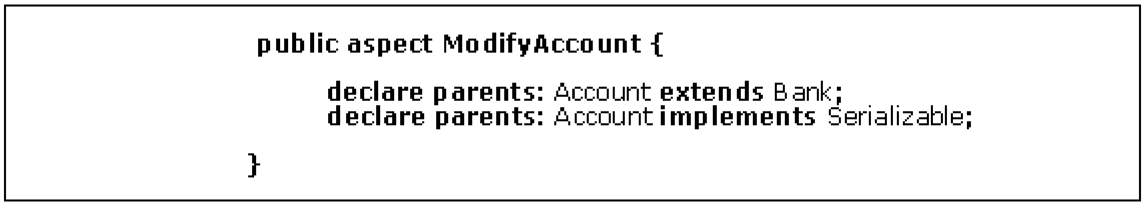


```
public aspect AddAccountID {
    private int Account.accountID = 0;
    public void Account.setAccountID(int id){
                accountID = id;
    }
    public int Account.getAccountID() {
                return accountID;
    }
}
```

Figure 4.5: Data member and method introduction

Modifying the class hierarchy

AspectJ also provides a construct (*declare parents*) for modifying the inheritance hierarchy of existing classes. This construct can declare new super-classes and super-interfaces for an existing class as demonstrated in figure 4.6. The *Account* class is now serializable and a subclass of *Bank*.



```
public aspect ModifyAccount {

      declare parents: Account extends Bank;
      declare parents: Account implements Serializable;

}
```

Figure 4.6: Modifying the hierarchy of a class using: *declare parents*

Aspect precedence

Multiple crosscutting behaviours may apply to the same join point in a system with multiple aspects. The outcome of an unspecified execution order of crosscutting behaviours at common join points is not only unpredictable; it may also be undesirable. AspectJ provides the *declare precedence* construct for controlling the execution order of advices at these common join points. This construct takes the following form:

declare precedence: TypePattern1, TypePattern2, …;

The aspects matching *Typepattern1* have a higher precedence than those matching *Typepattern2*, and so on. What this means from the viewpoint of an advice is: the *before* and *around* advice of higher precedence aspects executes before those of lower precedence aspects whereas the *after* advice of lower precedence aspects executes before those of higher precedence aspects.

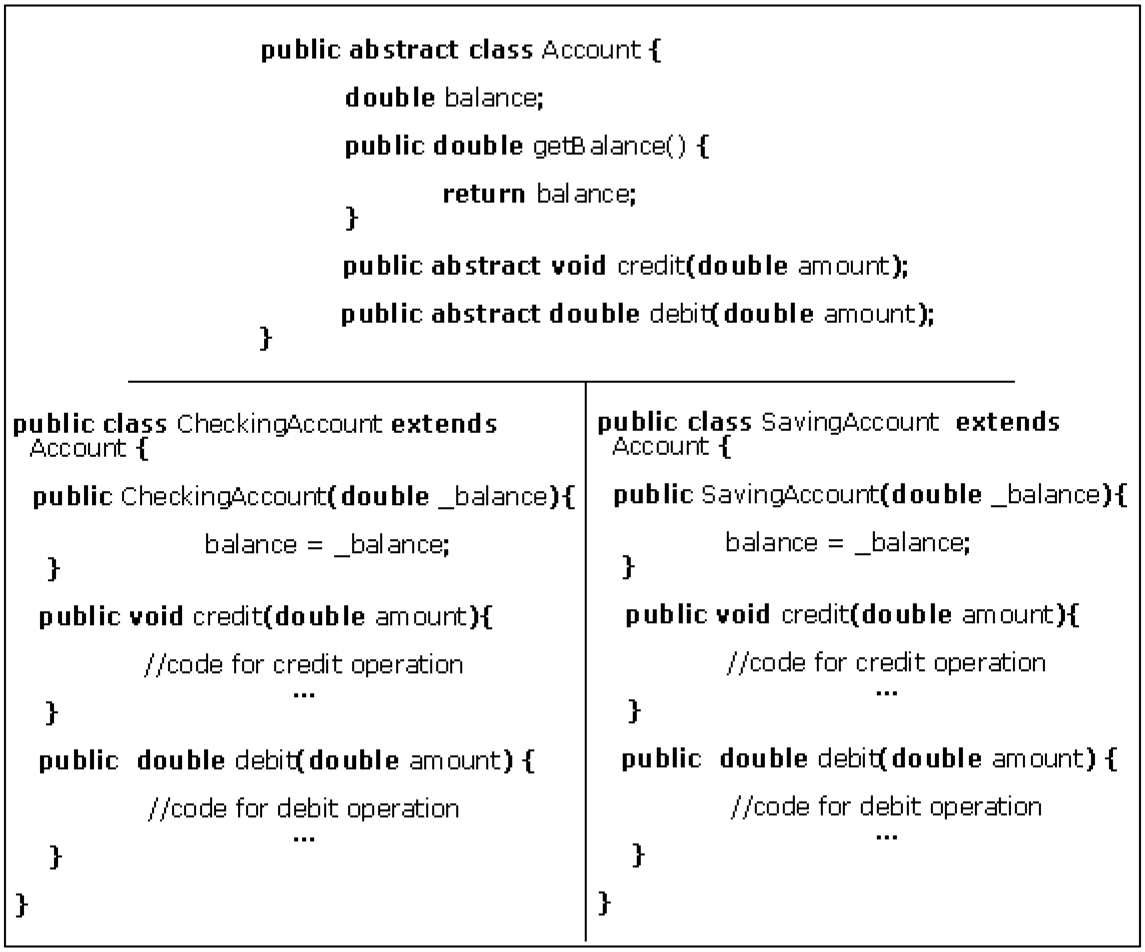
Chapter 5 ~ AspectJ Implementation of the ACID framework

I have introduced the foundational concepts (transactions, aspect-oriented programming, AspectJ and the case study) of this thesis in the preceding chapters. In this chapter, I present and critique potential AspectJ implementations for each of the aspects introduced in the case study. I also highlight the encountered implementation limitations for each aspect. In chapter 6, I provide an in-depth evaluation of these limitations, accompanied with potential AspectJ improvements.

Section 5.1 of this chapter introduces the sample base application used in this thesis. I introduce *abstract introductions* – a popular idiom for implementing reusable static crosscutting aspects - in section 5.2. A stripped-down implementation of the aspects and their limitations are discussed in section 5.3.

5.1 Sample Base Application

The sample application that will be used for demonstrating the implementation of reusable aspects is presented in figure 5.1. This is a simplified version of a typical account class and is intended just for demonstration purposes. The application consists of one abstract super class (*Account*) and two concrete subclasses (*SavingAccount* and *CheckingAccount*). The Account class has one field (*balance*), two abstract methods (*credit(...)* and *debit(...)*) and one non-abstract method (*getBalance()*). The subclasses provide a constructor and implementations for the inherited abstract methods but do not override the inherited non-abstract method (i.e., *getBalance()*).



```
public abstract class Account {
        double balance;
        public double getBalance() {
                return balance;
        }
        public abstract void credit(double amount);
        public abstract double debit(double amount);
}
```

```
public class CheckingAccount extends
  Account {
  public CheckingAccount(double _balance){
                balance = _balance;
  }
  public void credit(double amount){
          //code for credit operation
                    ...
  }
  public  double debit(double amount) {
          //code for debit operation
                    ...
  }
}
```

```
public class SavingAccount  extends
  Account {
  public SavingAccount(double _balance){
                balance = _balance;
  }
  public void credit(double amount){
          //code for credit operation
                    ...
  }
  public  double debit(double amount) {
          //code for debit operation
                    ...
  }
}
```

Figure 5.1: Sample base application

5.2 Reusability through the *Abstract Introduction* Idiom

As discussed in chapter 3 (the case study), the functionality of some aspects can only be achieved through the assistance provided by other aspects (e.g., *AutoRecoverable* depends on *Recoverable* and *AccessClassified*). Aspects therefore require an oblivious way for expressing their need of the functionalities (i.e., both *static* and *dynamic* crosscutting behaviours) provided by other aspects (i.e., *inter-aspect* *configurability*). In addition, these aspects should be generic enough to be used in different contexts besides ensuring the ACID properties of transactional objects. Hence, we also require an oblivious way for identifying the classes to which an aspect should be applied (i.e., *aspect-to-class binding*).

The *abstract introduction* idiom (also known as *indirect introduction*) [19,20] has been proposed as a strategy for implementing aspects that can be reused in different contexts. It allows us to "*collect several extrinsic properties from different perspectives within one unit and defers the binding to existing objects*" [19]. In other words, the target classes of the static and dynamic crosscutting behaviors are unknown until weave-time. This strategy has three participants (figure 5.2):

- *Introduction container*: a construct used as the target for the inter-type member declarations.
- *Introduction loader*: the aspect that introduces crosscutting behaviors and ancestors to the *introduction container*.
- *Container connector*: the aspect used for connecting the *introduction container* to the base application classes.

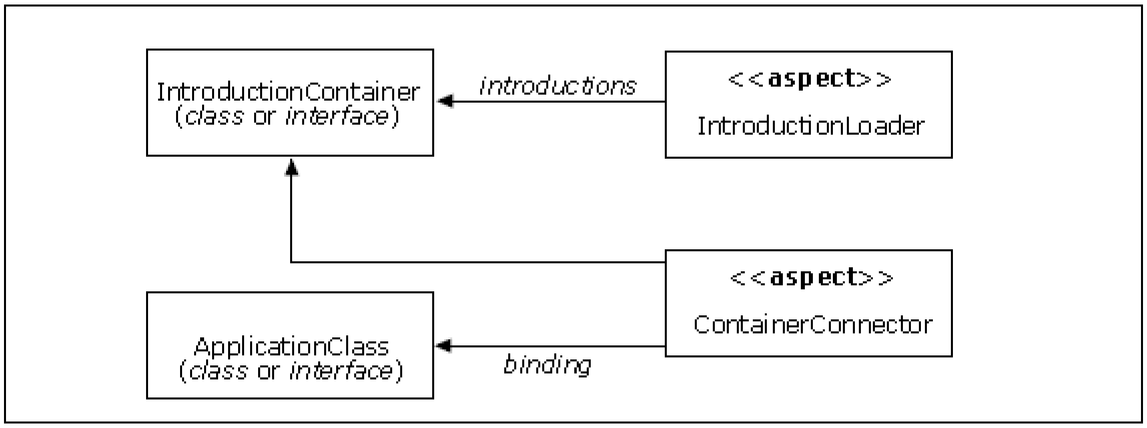
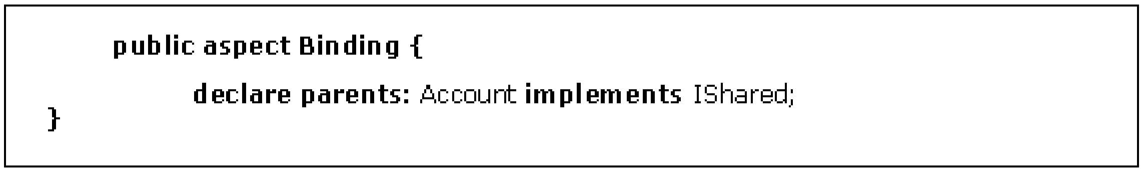


Figure 5.2: Abstract introduction

The *introduction container* serves a dual purpose in the context of this thesis. First, it enables the aspects (i.e., both *static* and *dynamic* crosscutting behaviours) to be reused in different contexts; second, it helps in identifying the classes to which the crosscutting behaviours of an aspect should be applied. The *introduction container* can either be a *class* or an *interface* in AspectJ. A *class* would not be an appropriate *introduction container* for our purpose because multiple-inheritance is not supported in Java and an aspect might rely on the functionalities of several other aspects. In addition, a *class* cannot be an ancestor to an *interface*. Consequently, interfaces are used as the *introduction container* for each of the aspects in this thesis.

I associate a dummy interface to each of the aspects identified in the case study. For instance, the interface *IShared* is associated to the aspect *Shared*, *IAutoRecoverable* is associated to *AutoRecoverable* and so on. Each aspect is then implemented to apply its functionality to all the classes that implements its associated interface (e.g., the *Shared* aspect is applied to all classes that implements the *IShared* interface). *Aspect-to-class binding* is typically achieved through the *declare parents* construct of AspectJ (figure 5.3) but explicit support for *inter-aspect configurability* is not yet supported. *Inter-aspect configurability* was achieved by having the associated interface of an aspect implement the interfaces of the aspects it depends on. For instance, inter-aspect configurability was achieved in the *AutoRecoverable* aspect by having *IAutoRecoverable* implement *IAccessClassified*, *IRecoverable*, (as demonstrated figure 5.4).


```
        public aspect Binding {
                declare parents: Account implements IShared;
  }
```
Figure 5.3: Aspect-to-class binding

```
public aspect AutoRecoverable {
    declare parents: IAutoRecoverable implements IRecoverable, IAccessClassified;
}
```

Figure 5.4: Inter-aspect configurability

Figure 5.3 brands the *Account* class as *IShared*; hence, the crosscutting behaviours of the *Shared* aspect would be applied to all the objects of the *Account* class at runtime. Figure 5.4 expresses the need of the *AutoRecoverable* aspect for the functionalities provided by the *Recoverable* and *AccessClassified* aspects (*see section 3.1.9*). Hence, an *AutoRecoverable* object is by default *Recoverable* and *AccessClassified*. Observe that the use of interfaces in achieving *aspect-to-class binding* and *inter-aspect configurability* satisfies the oblivious property as suggested in [21] – developers do not have to modify their systems to accommodate these aspects and the base application is completely ignorant of their existence.

5.3 Implementing the ACID Framework

I discuss and critique the implementations of each of the aspects identified in the case study in this section. Encountered AspectJ limitations are also highlighted but will be addressed in detail in chapter 6.

Be reminded that an object in an AspectJ environment has three types of methods: those *inherited* from super-classes and super-interfaces, those *declared* by the class, and those *introduced* through direct or indirect introductions (as opposed to just *inherited* and *declared* methods in Java). The same is true for data members. An AOP transaction framework must therefore recognize this new dimension and handle it accordingly. These implementations also assume that the interactions between the transaction framework and the transactional objects occur through well-defined public interfaces. Consequently, the enforcement of the ACID properties of transactions is addressed only at this level.

5.3.1 AccessClassified

Summary of functionality

- This aspect classifies every public method (*inherited, declared, introduced*) of an object as either a *read*, a *write* or an *update* operation.

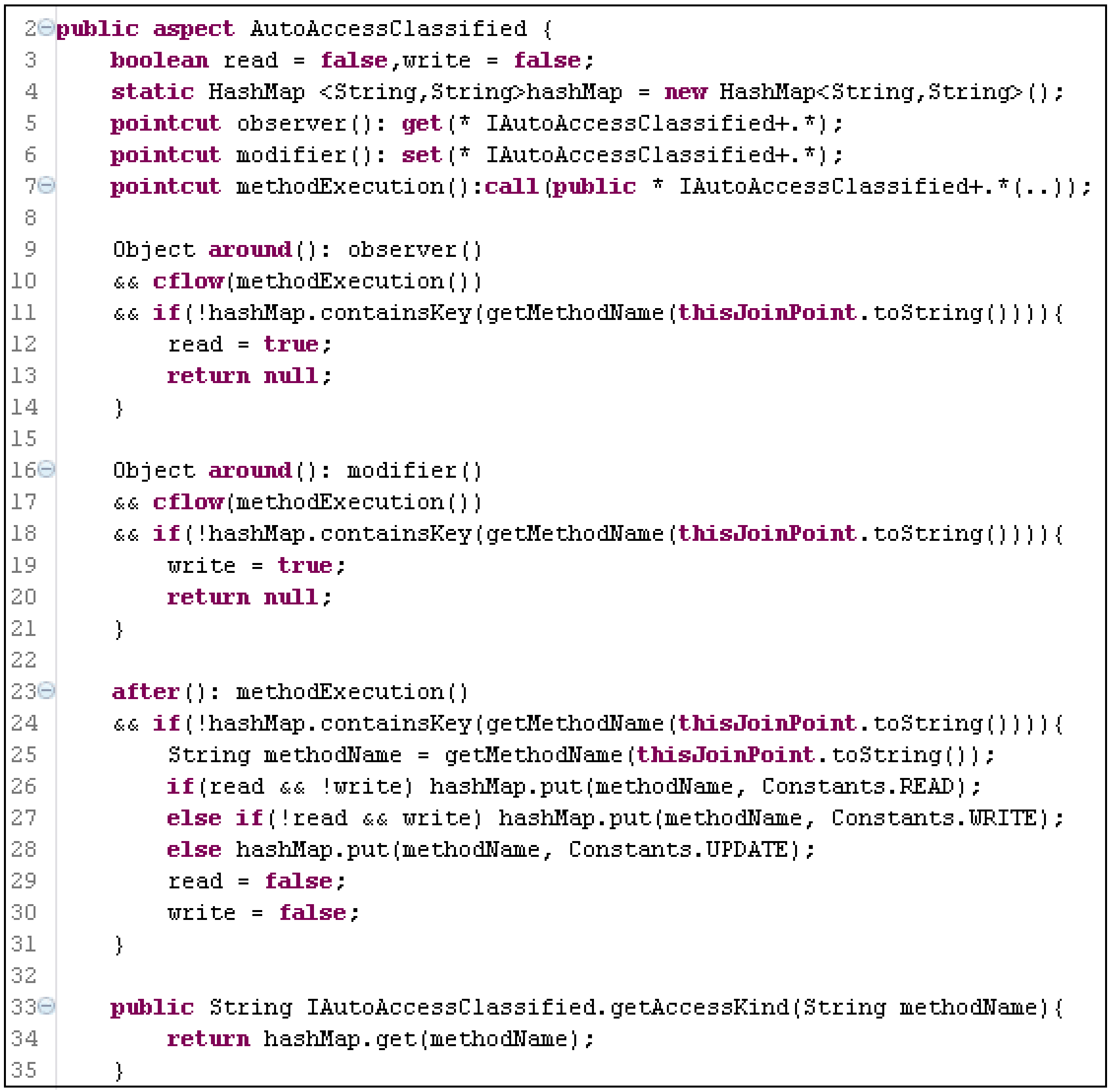
Implementation

Option-1: Implementation

The effectiveness of an ACID transactional framework is conditional on its ability to accurately identify the access type of the operations of a transactional object. An unnoticeable erroneous classification might place the system in an inconsistent state - with potential costly consequences. An ideal solution would therefore involve an automation of the method classification process instead of relying on developers.

An automated solution would have to anticipate all field references within the control flow of a given operation and classify the operation accordingly. An operation with only *get* field references in its control flow would be classified as a *read*, an operation with only *set* field references in its control flow would be classified as a *write*, and an operation with both *get* and *set* field references in its control flow would be classified as an *update*.

AspectJ provides two pointcuts for capturing *get* and *set* field references: *get(FieldPattern)* and *set(FieldPattern)*. Figure 5.5 illustrates how these pointcuts can be used in automating the operation classification process.



```
2  public aspect AutoAccessClassified {
3      boolean read = false,write = false;
4      static HashMap <String,String>hashMap = new HashMap<String,String>();
5      pointcut observer(): get(* IAutoAccessClassified+.*);
6      pointcut modifier(): set(* IAutoAccessClassified+.*);
7      pointcut methodExecution():call(public * IAutoAccessClassified+.*(..));
8
9      Object around(): observer()
10     && cflow(methodExecution())
11     && if(!hashMap.containsKey(getMethodName(thisJoinPoint.toString()))){
12         read = true;
13         return null;
14     }
15
16     Object around(): modifier()
17     && cflow(methodExecution())
18     && if(!hashMap.containsKey(getMethodName(thisJoinPoint.toString()))){
19         write = true;
20         return null;
21     }
22
23     after(): methodExecution()
24     && if(!hashMap.containsKey(getMethodName(thisJoinPoint.toString()))){
25         String methodName = getMethodName(thisJoinPoint.toString());
26         if(read && !write) hashMap.put(methodName, Constants.READ);
27         else if(!read && write) hashMap.put(methodName, Constants.WRITE);
28         else hashMap.put(methodName, Constants.UPDATE);
29         read = false;
30         write = false;
31     }
32
33     public String IAutoAccessClassified.getAccessKind(String methodName){
34         return hashMap.get(methodName);
35     }
```

Figure 5.5: Automatic classification of operations

The *get* and *set* pointcuts for capturing field references on *IAutoAccessClassified* objects are defined on lines 5 and 6 respectively, line 7 defines a pointcut for capturing public calls to *IAutoAccessClassified* objects. The *around* advice (lines 9 to 21) intercepts and classifies all field references in the control flow of the current executing public operation with target *IAutoAccessClassified*. The *getMethodName(**thisJoinPoint**.toString())* function is a helper method for obtaining the name of the current executing method from the

thisJoinPoint aspect variable. Observe that there is no *proceed()* statement in either of the *around* advices (i.e., the captured join points are not executed). The *after* advice (lines 23 to 31) then determines the access type of the operation based on the kinds of field references in its control flow and stores this information in a HashMap. The *getAccessKind(String methodName)* method (lines 33 to 35) returns the access type of the method with name *methodName* on the *IAutoAccessClassified* object.

Option-1: Discussion

Although this implementation successfully automates the operation classification process, it has four significant drawbacks. First, every operation must be executed twice (although invoked only once by the developer): the first time to determine its access type (*classification phase*) and the second time, to actually execute the operation by reflection (*execution phase*). Additional aspects are obviously required to mask this complexity from the developer. Theoretically, the execution time of an operation would double the first time it is executed in the presence of automatic classification.

Secondly, conditional state modification statements (e.g., set field references within an *if-statement*) within an operation may not be executed the first time it executes. This implies that an *update* operation may be erroneously classified as a *read* operation. There is therefore the potential of corrupting the state of an object since subsequent requests for an operation’s access type is obtained from the hash map.

Thirdly, the field reference pointcuts (lines 5 and 6) captures accesses to the fields of transactional objects only. This implies that set field references on non-transactional objects in the control flow of transactional operations will always be executed twice - during the *classification phase* and the *execution phase* – putting the system in an inconsistent state. Our implementation assumes that transactional operations do not access non-transactional objects.

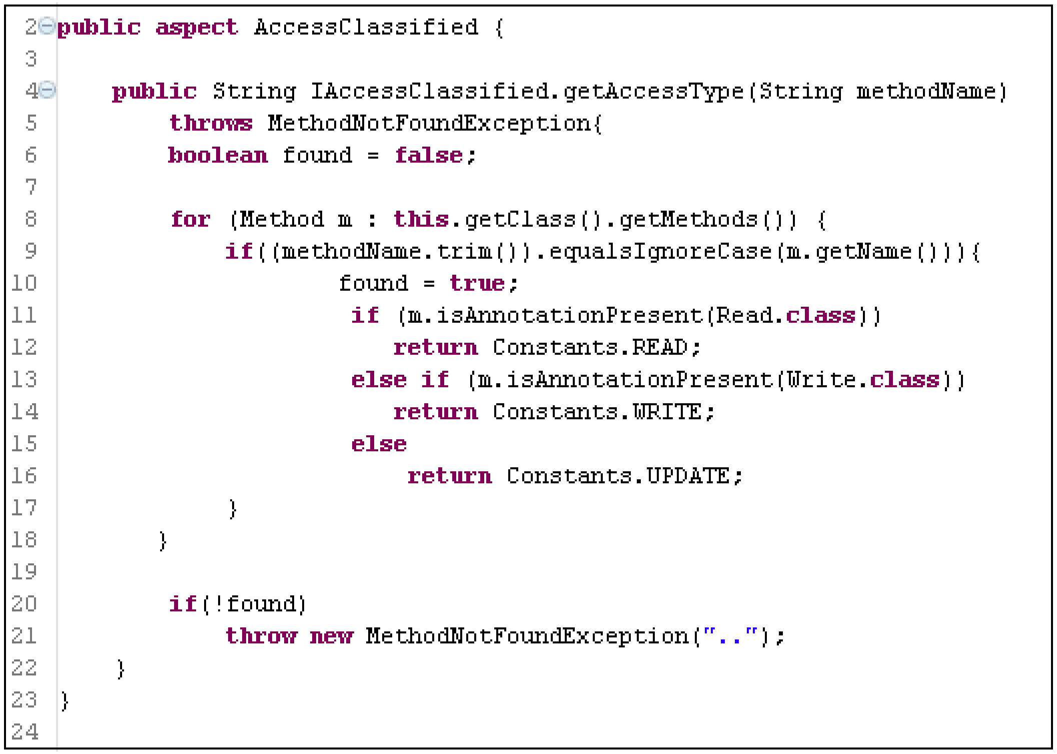
Finally, this solution relies on the *if(Boolean)* pointcut (lines 11, 18 and 24) to avoid classifying an operation every time it is invoked. That is, operations are only classified once and their access type subsequently obtained from the HashMap. This implies that an operation call will always be intercepted by the pointcuts but not necessarily classified. The performance overhead introduced by the *if(Boolean)* pointcut cannot be significantly reduced at runtime since AspectJ does not currently support runtime disabling and re-enabling of pointcuts (*AspectJ limitation*). Ideally, all the pointcuts of the *AutoAccessClassified* aspect should be disabled as soon as the access type of all the public operations is known. The performance overhead introduced by this deficiency, combined with the need for dual operation execution might outweigh the benefits of automatic classification.

The runtime performance overhead may be significantly reduced or even eliminated if the classification process is performed at compile-time. Some AOP compilers already support the identification of *get* and *set* field references in the control flow of a method at compile-time. Such compilers may therefore be optimized to use this information in classifying and annotating the methods of a class with metadata at compile-time. However, this is out of the scope of this thesis.

Option-2: Implementation

This option places the burden of classifying the operations of a transactional object on the developer. Developers are required to classify every public method (*inherited, declared, introduced*) of a transactional object as either a *read*, a *write* or an *update* using the marker annotations introduced in section 4.1. The *AccessClassified* aspect (figure 5.6) introduces a method (*getAccessType(String methodName)*) to every *IAccessClassified* object that examines these annotations by reflection at runtime and classify each operation accordingly.

Line 8 obtains all the methods (*inherited, declared, introduced*) of the object, lines 9 to 18 determines the annotation type associated with the method of interest using the *Method.isAnnotationPresent(..)* method of the Java reflection framework.



```
 2 public aspect AccessClassified {
 3
 4     public String IAccessClassified.getAccessType(String methodName)
 5         throws MethodNotFoundException{
 6         boolean found = false;
 7
 8         for (Method m : this.getClass().getMethods()) {
 9             if((methodName.trim()).equalsIgnoreCase(m.getName())){
10                     found = true;
11                     if (m.isAnnotationPresent(Read.class))
12                        return Constants.READ;
13                     else if (m.isAnnotationPresent(Write.class))
14                        return Constants.WRITE;
15                     else
16                         return Constants.UPDATE;
17             }
18         }
19
20         if(!found)
21             throw new MethodNotFoundException("..");
22     }
23 }
24
```

Figure 5.6: Manual classification of operations

AspectJ currently supports *direct introduction* of annotated methods (i.e., *@Read int Account.getAccountID()*) into a target class. These methods will therefore be appropriately classified. However, *Indirect introduction* of annotated methods (i.e., introducing *@Read int getAccountID()* into the interface *IExample* - *@Read int IExample.getAccountID()* - and later binding the *Account* class to *IExample* as illustrated in section 5.2) into a target class is not yet supported in AspectJ. That is, the *getAccountID()* method will be introduced into the target interface (*IExample*) without the annotation (*@Read*) and later bound to the target application (*Account*) without the annotation (*AspectJ limitation*). The framework must therefore make worst-case assumptions (line 16) when such methods are encountered to ensure system consistency.

Option-2: Discussion

This option also has two significant drawbacks. First, its reliance on the developer in classifying the methods of transactional objects is a time bomb with potential costly consequences. Also, developers may ignore or forget to classify the methods of transactional objects, resulting in a system where all the operations are considered conflicting. This may greatly affect the performance of such a system. Second, *observer* operations introduced by *indirect introduction* are by default classified as modifiers – another performance concern.

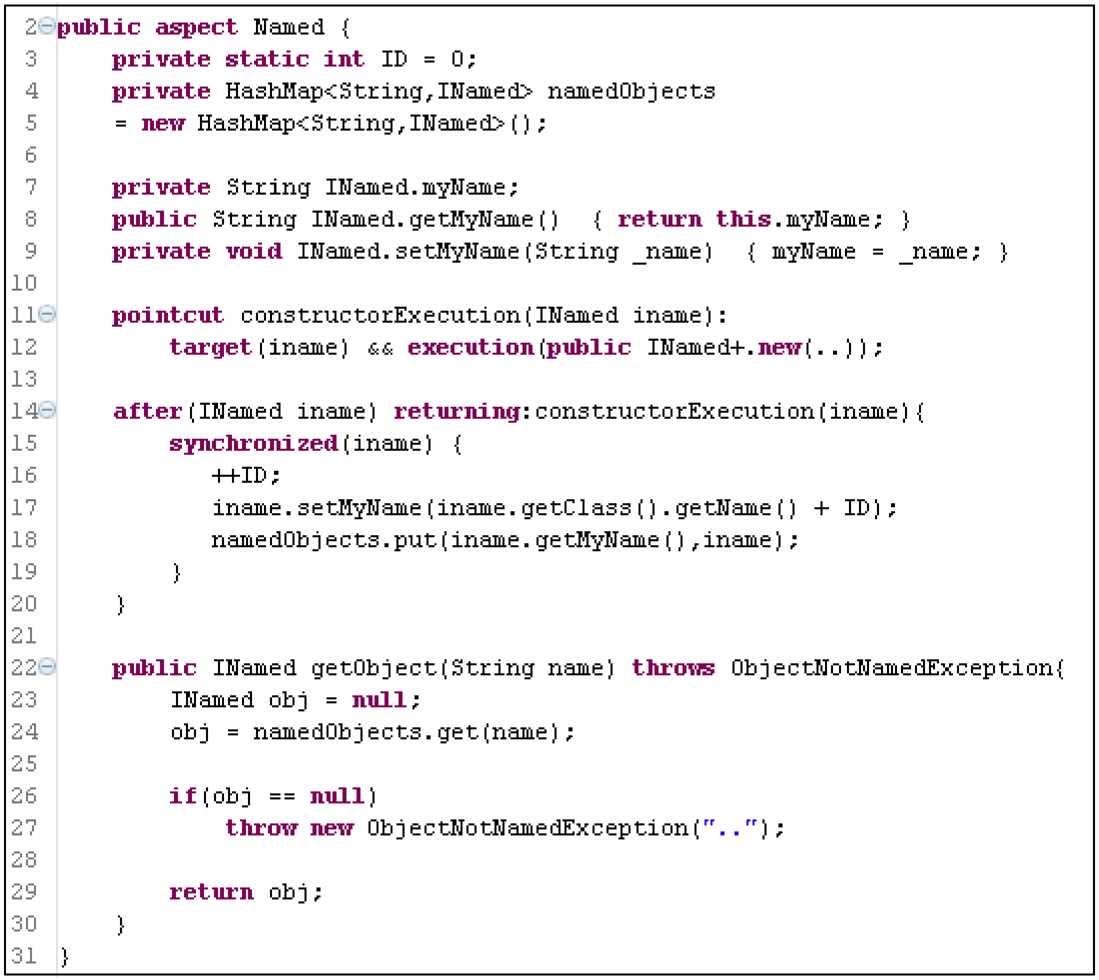
Ideally, the correctness of the classification should be verified at compile time and the appropriate warning/error message emitted. For instance, there shouldn't be any *set* field references in the control flow of a method annotated as *read*. This cannot be accomplished with the *declare error* and *declare warning* constructs of AspectJ because the *cflow()* pointcut is not statically determinable. Our ideal AOP tool should provide a pointcut that supports such functionality at compile-time.

5.3.2 Named

Summary of functionality

- Every object creation operation must associate a unique name to the object that is created.
- Every named object knows its name and it should be possible to retrieve an object given its name.
- An object's name must remain unchanged throughout its lifetime.

Implementation



```
public aspect Named {
    private static int ID = 0;
    private HashMap<String,INamed> namedObjects
    = new HashMap<String,INamed>();

    private String INamed.myName;
    public String INamed.getMyName()  { return this.myName; }
    private void INamed.setMyName(String _name)  { myName = _name; }

    pointcut constructorExecution(INamed iname):
        target(iname) && execution(public INamed+.new(..));

    after(INamed iname) returning:constructorExecution(iname){
        synchronized(iname) {
           ++ID;
           iname.setMyName(iname.getClass().getName() + ID);
           namedObjects.put(iname.getMyName(),iname);
        }
    }

    public INamed getObject(String name) throws ObjectNotNamedException{
        INamed obj = null;
        obj = namedObjects.get(name);

        if(obj == null)
            throw new ObjectNotNamedException("..");

        return obj;
    }
}
```

Figure 5.7: Implementation of the Named aspect

Figure 5.7 illustrates a possible implementation of the *Named* aspect. Line 7 introduces a private field named *myName* into the object that the aspect is applied to, line 8 introduces a public method (*getMyName()*) for retrieving the object's name, and line 9 introduces a private method (*setMyName(String)*) for setting the object's name. The private method *setMyName(String)* is accessible only within this aspect (i.e., it cannot be accessed even by the base application to which the aspect is bound). Hence, once initialized, the object's name cannot be changed.

The pointcut for picking out the constructor execution of an *INamed* object is defined on lines 11 to 12, and lines 14 to 20 defines an *after* advice for setting the object's name. The object's name is the fully qualified name the class name (i.e., the full package path) plus a unique ID (line 15) and therefore unique. Lines 22 to 30 defines a method (*getObject()*) for retrieving an object given its name.

Discussion

As demonstrated, reusable implementations may require the introduction of additional information (data members and methods) into their target classes to facilitate their functionality. With several aspects in a system, there is the possibility of introducing methods or fields with the same name into the target class - resulting in naming conflicts. An alternative implementation is to keep the additional information within the aspect and associate an instance of the aspect state to every instance of an *INamed* object. This sounds good, except that the aspect-to-object association must occur at a captured join point but purely static crosscutting aspects (such as the manual *AccessClassified* aspect – figure 5.6) do not contain join points. Consequently, the name conflict issue is unavoidable (*AspectJ limitation*).

Another deficiency worth nothing is the inability of the *set* pointcut of AspectJ in capturing reflective field references; hence, reflective changes of an object's name cannot be prevented. Why not just declare the name field final to prevent reflective changes – you ask? First, because a final field can only be initialized within a constructor and the constructor execution pointcut (lines 10 to 11) does not satisfy this requirement even though its join point occurs within the body of the constructor (see section 4.2.1). Secondly, indirect initialization of a final field using a set-field method from within an advice that picks out a constructor execution (such as in lines 13 to 17) is also not feasible.

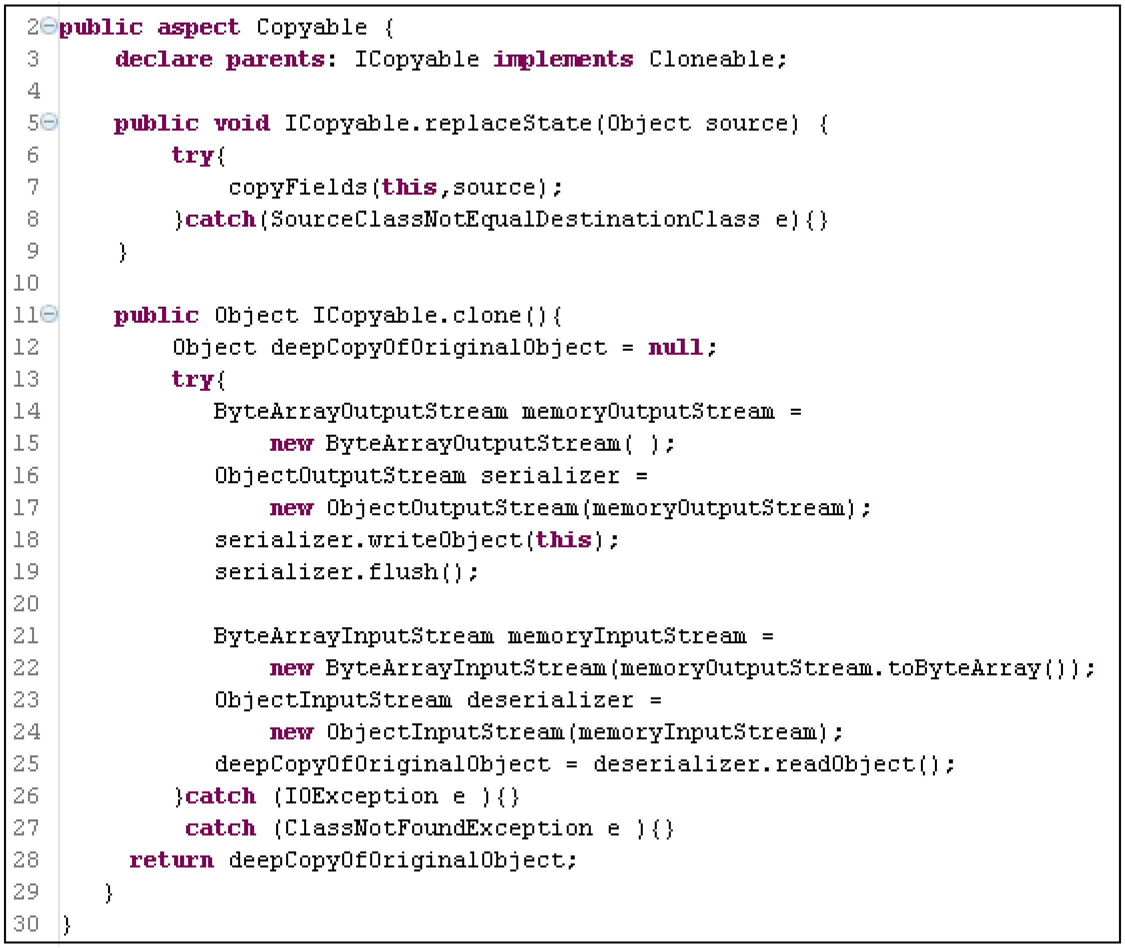
5.3.3 Copyable

Summary of functionality

- An *ICopyable* object is *Cloneable*, i.e., this aspect enables an object to create an identical copy of itself.
- An *ICopyable* object can also replace its state with that of another object of the same class.

Implementation

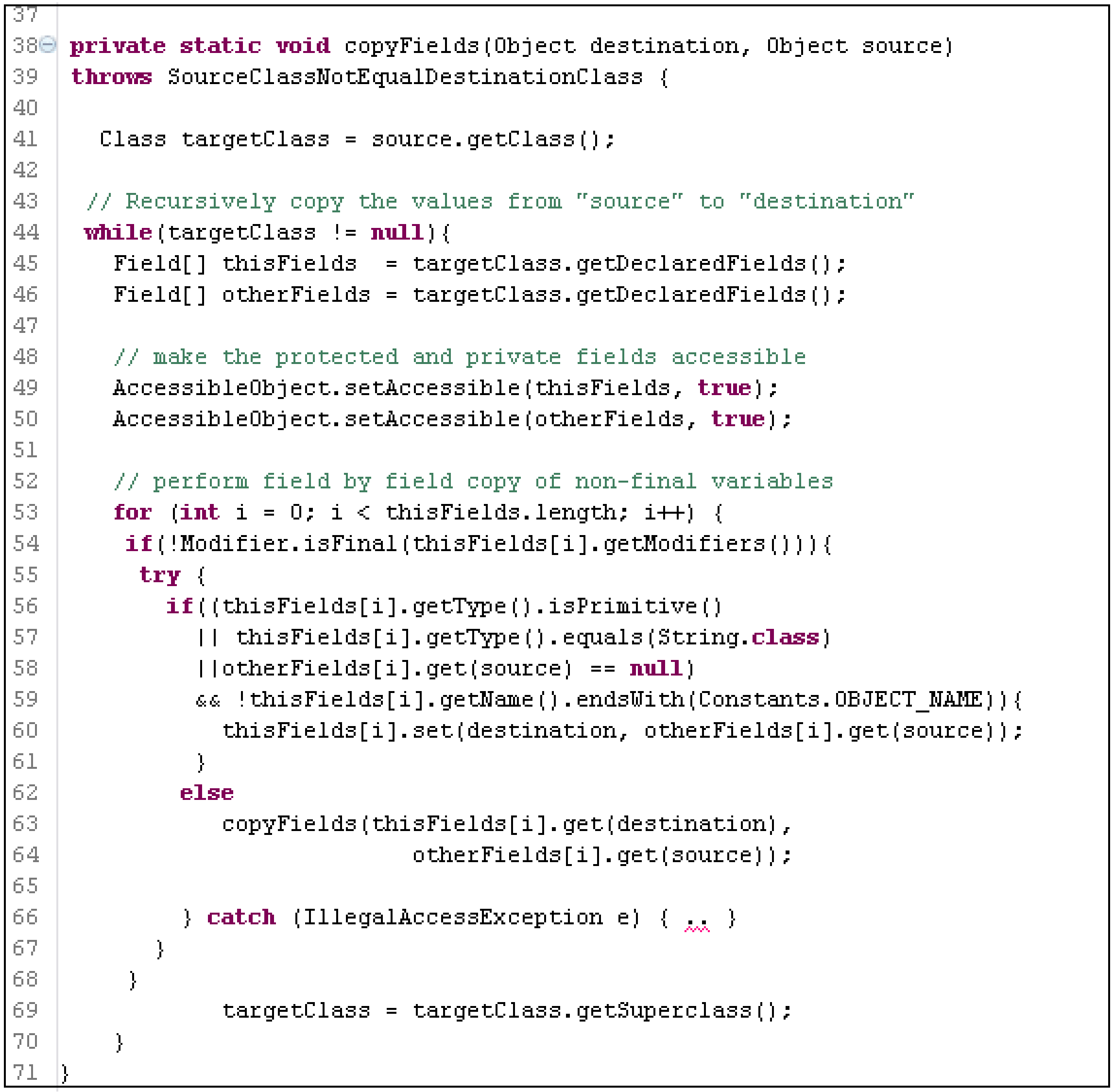
The *Copyable* aspect (figure 5.8) introduces static crosscutting behaviours to all classes that implement the *ICopyable* interface to facilitate the state replacement and cloning functionality. Line 3 enables all classes that implement the *ICopyable* interface to be *Cloneable*. Lines 11 to 29 present one possible implementation of the clone method. This implementation requires the target class (including its composed classes) to be serializable so as to support deep cloning. The decision of whether to support *deep* or *shallow cloning* of an object's state is application dependent - each having its strengths and weaknesses - and certainly not an AspectJ issue. This cloning strategy is therefore intended just for demonstration purposes. Developers can seamlessly override this cloning implementation by providing a custom *clone()* implementation within classes that implement the *ICopyable* interface.



```
 2 public aspect Copyable {
 3     declare parents: ICopyable implements Cloneable;
 4 
 5     public void ICopyable.replaceState(Object source) {
 6         try{
 7             copyFields(this,source);
 8         }catch(SourceClassNotEqualDestinationClass e){}
 9     }
10 
11     public Object ICopyable.clone(){
12         Object deepCopyOfOriginalObject = null;
13         try{
14            ByteArrayOutputStream memoryOutputStream =
15                new ByteArrayOutputStream( );
16            ObjectOutputStream serializer =
17                new ObjectOutputStream(memoryOutputStream);
18            serializer.writeObject(this);
19            serializer.flush();
20 
21            ByteArrayInputStream memoryInputStream =
22                new ByteArrayInputStream(memoryOutputStream.toByteArray());
23            ObjectInputStream deserializer =
24                new ObjectInputStream(memoryInputStream);
25            deepCopyOfOriginalObject = deserializer.readObject();
26         }catch (IOException e ){}
27          catch (ClassNotFoundException e ){}
28      return deepCopyOfOriginalObject;
29    }
30 }
```

Figure 5.8: Implementation of the Copyable aspect – Part I

Lines 5 to 9 provide the method (*replaceState(Object)*) for replacing the state of one object with that of another object of the same class. The *copyFields(this, source)* method – figure 5.9 - performs a field-by-field copy of the state (*inherited*, *declared*, *introduced* and those of *composed* objects) of the source object to the invoking object.



```
37
38 private static void copyFields(Object destination, Object source)
39 throws SourceClassNotEqualDestinationClass {
40
41   Class targetClass = source.getClass();
42
43   // Recursively copy the values from "source" to "destination"
44   while(targetClass != null){
45     Field[] thisFields  = targetClass.getDeclaredFields();
46     Field[] otherFields = targetClass.getDeclaredFields();
47
48     // make the protected and private fields accessible
49     AccessibleObject.setAccessible(thisFields, true);
50     AccessibleObject.setAccessible(otherFields, true);
51
52     // perform field by field copy of non-final variables
53     for (int i = 0; i < thisFields.length; i++) {
54      if(!Modifier.isFinal(thisFields[i].getModifiers())){
55       try {
56         if((thisFields[i].getType().isPrimitive()
57           || thisFields[i].getType().equals(String.class)
58           ||otherFields[i].get(source) == null)
59           && !thisFields[i].getName().endsWith(Constants.OBJECT_NAME)){
60             thisFields[i].set(destination, otherFields[i].get(source));
61           }
62          else
63             copyFields(thisFields[i].get(destination),
64                           otherFields[i].get(source));
65
66          } catch (IllegalAccessException e) { ... }
67        }
68      }
69             targetClass = targetClass.getSuperclass();
70     }
71 }
```

Figure 5.9: Implementation of the Copyable aspect – Part II

Discussion

Fields declared as *final* cannot be replaced - even by reflection. The state replacement method cannot therefore replace the value of a final field. This implementation also detects the presence of the *Named* aspect to avoid modifying the identity (*Constants.OBJECT_NAME – line 59*) of an *INamed* object.

5.3.4 Shared

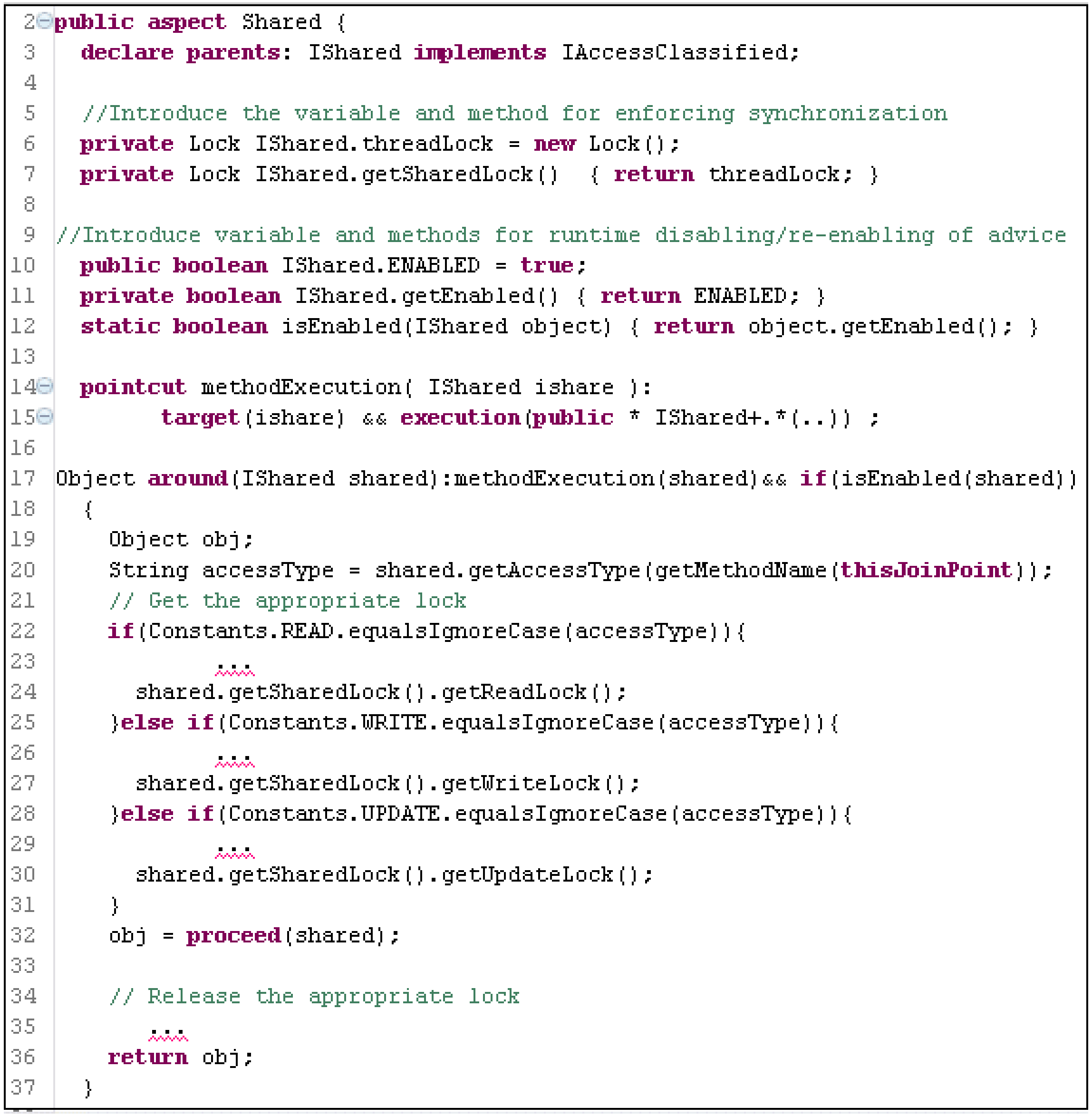
Summary of functionality

- This aspect obligates a transaction to obtain a *read*, *write*, or an *update* lock before executing a method (*inherited*, *declared* and *introduced*) of an *IShared* object.
- Previously acquired locks are released when returning from the method invocation.

Implementation

The implementation of the *Shared* aspect is presented in figure 5.10. The *Shared* aspect expresses its need for the functionality provided by *AccessClassified* on line 3, lines 6 to 7 introduce the static crosscutting behaviours for synchronizing access to an object's state, and lines 10 to 12 introduce fields and methods for supporting runtime enabling or disabling of advices on a per-object basis (e.g., for disabling the *Shared* advice on the objects in the history of committed states). This aspect relies on *AccessClassified* to distinguish *observe*r operations from *modifier* operations.

The *around* advice (lines 17 to 37) obligates every thread executing an operation (with the exception of static methods because they do not have a *this* reference) on a *Shared* object to acquire the appropriate lock (lines 21 to 31) before proceeding (line 32). The previously acquired lock is then released (lines 34 to 35) after the operation is executed. A well-known problem with lock-based solutions is the possibility of deadlock. For instance, deadlock can occur in this implementation if a thread with a read lock on an object wants to execute another method on the same object that requires a write lock without releasing its read lock. This implementation allows a thread to upgrade its lock in such instances in a deadlock-free manner (the code has been edited out for clarity).



```
 2 public aspect Shared {
 3   declare parents: IShared implements IAccessClassified;
 4
 5   //Introduce the variable and method for enforcing synchronization
 6   private Lock IShared.threadLock = new Lock();
 7   private Lock IShared.getSharedLock()  { return threadLock; }
 8
 9 //Introduce variable and methods for runtime disabling/re-enabling of advice
10   public boolean IShared.ENABLED = true;
11   private boolean IShared.getEnabled() { return ENABLED; }
12   static boolean isEnabled(IShared object) { return object.getEnabled(); }
13
14   pointcut methodExecution( IShared ishare ):
15         target(ishare) && execution(public * IShared+.*(..)) ;
16
17 Object around(IShared shared):methodExecution(shared)&& if(isEnabled(shared))
18   {
19     Object obj;
20     String accessType = shared.getAccessType(getMethodName(thisJoinPoint));
21     // Get the appropriate lock
22     if(Constants.READ.equalsIgnoreCase(accessType)){
23             ...
24       shared.getSharedLock().getReadLock();
25     }else if(Constants.WRITE.equalsIgnoreCase(accessType)){
26             ...
27       shared.getSharedLock().getWriteLock();
28     }else if(Constants.UPDATE.equalsIgnoreCase(accessType)){
29             ...
30       shared.getSharedLock().getUpdateLock();
31     }
32     obj = proceed(shared);
33
34     // Release the appropriate lock
35        ...
36     return obj;
37   }
```

Figure 5.10: Implementation of the Shared aspect

Discussion

This implementation has three significant drawbacks. First, as explained in section 3.2.2 (Multi-version lock based concurrency control), there is no need of the *Shared* aspect on objects that are accessed only by *observer* operations. At best, AspectJ supports only the disabling of advices through the *if(BooleanExpression)* pointcut (line 18). This implies that the *target-joinpoint* (lines 14 to 15) will always be intercepted but the execution of its associated advice is conditional on the value of *BooleanExpression*. Consequently, every operation call to the objects in the history of committed states will unnecessarily be intercepted - incurring an unnecessary performance overhead (*AspectJ limitation*). Disabling the pointcut(s) that intercept *target-joinpoint(s)* would eliminate the unnecessary interception of operation calls. Our ideal AOP tool should therefore provide support for runtime disabling and re-enabling of pointcuts so as to optimize system performance.

Secondly, a method call pointcut does not capture reflective calls to operations on *Shared* objects, making it impossible to detect and prevent the concurrent execution of conflicting operations. This is a deliberate decision made by the AspectJ team not to "*delve into the Java reflection library to implement call semantics*" [9]. Developers are advised to use the method execution pointcut (*execution(public * IShared+.*(..))*) instead. The method execution pointcut on the other hand does not pick out the execution (reflective or non-reflective) of non-overridden inherited methods when the target is the subclass. For instance, assuming that *CheckingAccount* is *IShared*, the execution of *CheckingAccount.getBalance()* is not captured by the aforementioned method execution pointcut; firstly, because the *getBalance()* method is not overridden in the *CheckingAccount* subclass; and secondly, because the execution join point of *getBalance()* occurs in the *Account* super class. Making the super class (i.e., *Account*) *IShared* would solve the problem but at the expense of obligating all its subclasses (e.g., *SavingAccount*) to also be *IShared*. Composing *call(public * IShared+.*(..))*) and *execution(public * IShared+.*(..))*) with an OR operator is also not a feasible solution because reflective invocations of *getBalance()* will not be captured. Developers must therefore decide between exploiting the benefits of inheritance (i.e., by not unnecessarily overriding inherited implementations) or capturing reflective method executions but not both (*Reflection/Super-class method execution dilemma*) – *AspectJ limitation*.

Finally, transactional objects by nature assume multiple roles as they move through the execution of a transaction. For instance, if the *SavingAccount* class implements both *ICopyable* and *IShared*, then *SavingAccount* objects would be treated as *ICopyable* from the perspective of the *Copyable* aspect and as *IShared* from the perspective of the *Shared* aspect. However, the *Shared* aspect must intercept every public method call (*inherited*, *declared* and *introduced*) on a *SavingAccount* object; including those introduced by *Copyable* (i.e., *replaceState()* and *clone()*). It is logical to assume that the call and execution of *SavingAccount.replaceState()* would be captured by both (*call(public * IShared+.*(..))*) and (*execution(public * IShared+.*(..))*) respectively since the method *replaceState()* was actually introduced into the *SavingAccount* class. This is however not the case because AspectJ associates the call and execution join point of *replaceState()* with *ICopyable*, instead of *SavingAccount* (*Weak aspect-to-class binding*). Therefore, there is the possibility of cloning or replacing the state of a *Shared* object while it is being modified by a concurrent transaction. The use of *indirect introduction* as a means for providing default interface implementation although effective, is therefore not equivalent to its Java counterpart (i.e., implementing the methods inherited from super interfaces within the subclasses) that actually associates these join points with *SavingAccount* – *AspectJ limitation*.

5.3.5 Serializeable

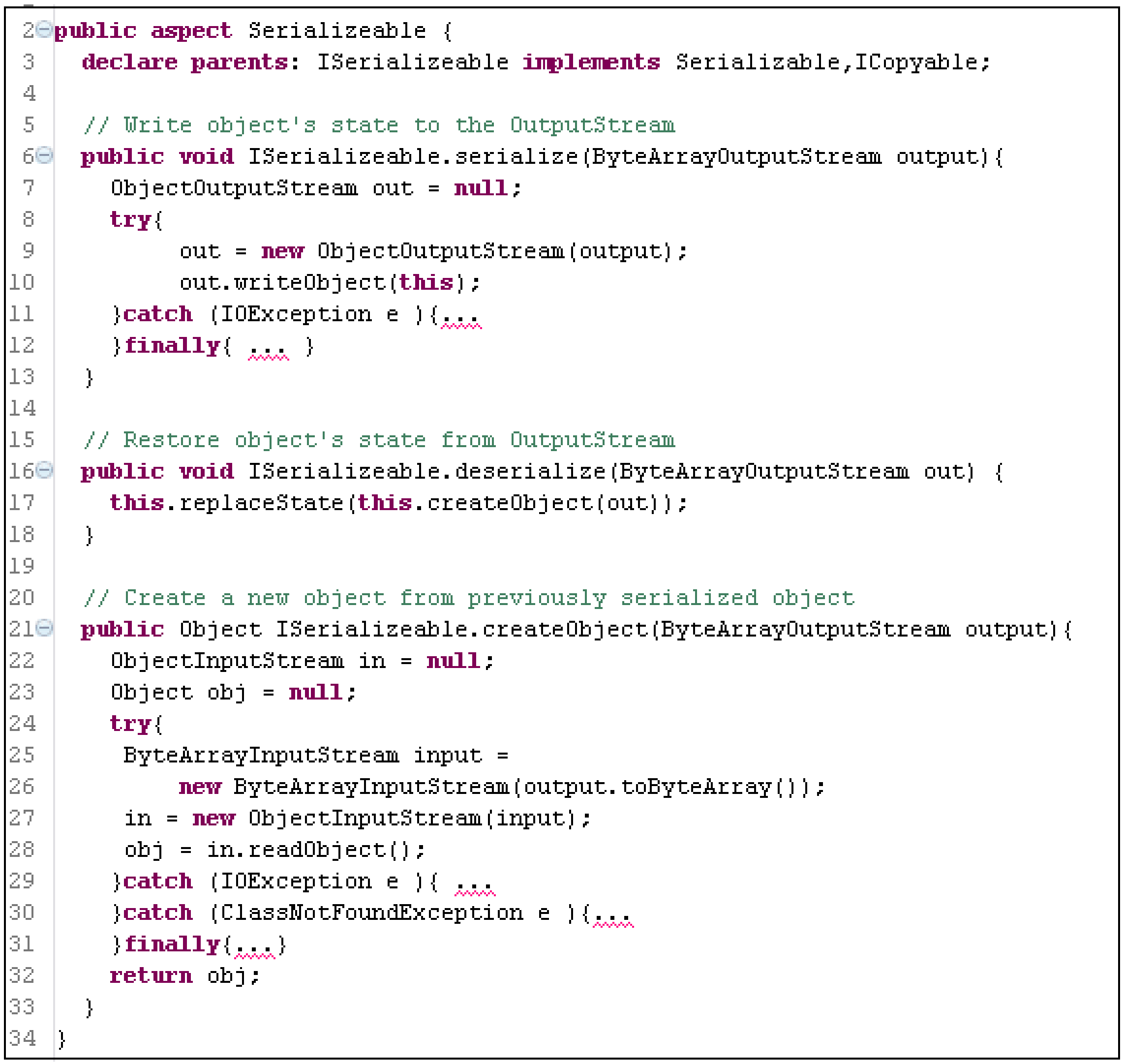
Summary of functionality

- An *ISerializeable* object knows how to write its state (*declared*, *inherited* and *introduced*) to a backend, restore its state by reading it from the backend and create a new object - initialized with the state read from the backend.

Implementation

The implementation of the *Serializeable* aspect is presented in figure 5.11. As with *Copyable*, this aspect introduces only static crosscutting behaviours into the target class to support the desired functionality. The backend in this case is an *OutputStream* and it is assumed that the object is serializable (i.e., implements *java.io.Serializable*) – line 3. As before, this is not an AspectJ issue and this implementation is intended just for demonstration purposes.

The *Serializeable* aspect expresses its need for the functionality provided by the *Copyable* aspects on line 3. The *Copyable* aspect is required for replacing the state of an object with its backend state. Lines 6 to 13 introduce the method (*serialize(..)*) for writing an object's state to an *OutputStream*, lines 16 to 18 introduce the method (*deserialize(..)*) for restoring an object's state from the backend, and lines 21 to 33 introduces the method (*createObject(..)*) for creating a new object from a previously saved state. This implementation expects each transaction to specify a unique *OutputStream* for serializing its perspective of the state of a transactional object.



```
public aspect Serializeable {
  declare parents: ISerializeable implements Serializable,ICopyable;

  // Write object's state to the OutputStream
  public void ISerializeable.serialize(ByteArrayOutputStream output){
    ObjectOutputStream out = null;
    try{
          out = new ObjectOutputStream(output);
          out.writeObject(this);
    }catch (IOException e ){...
    }finally{ ... }
  }

  // Restore object's state from OutputStream
  public void ISerializeable.deserialize(ByteArrayOutputStream out) {
    this.replaceState(this.createObject(out));
  }

  // Create a new object from previously serialized object
  public Object ISerializeable.createObject(ByteArrayOutputStream output){
    ObjectInputStream in = null;
    Object obj = null;
    try{
     ByteArrayInputStream input =
          new ByteArrayInputStream(output.toByteArray());
     in = new ObjectInputStream(input);
     obj = in.readObject();
    }catch (IOException e ){ ...
    }catch (ClassNotFoundException e ){...
    }finally{...}
    return obj;
  }
}
```

Figure 5.11: Implementation of the Serializeable aspect

Discussion

An ideal *Serializeable* aspect should support multiple backend representation formats (e.g., a database, a file or remote machine) and a mechanism for selecting a desired format. It is my opinion that an implementation of such an aspect will not reveal any additional AspectJ limitation. Consequently, my implementation of the *Serializeable* aspect is limited to *OutputStream* only.

5.3.6 Versioned

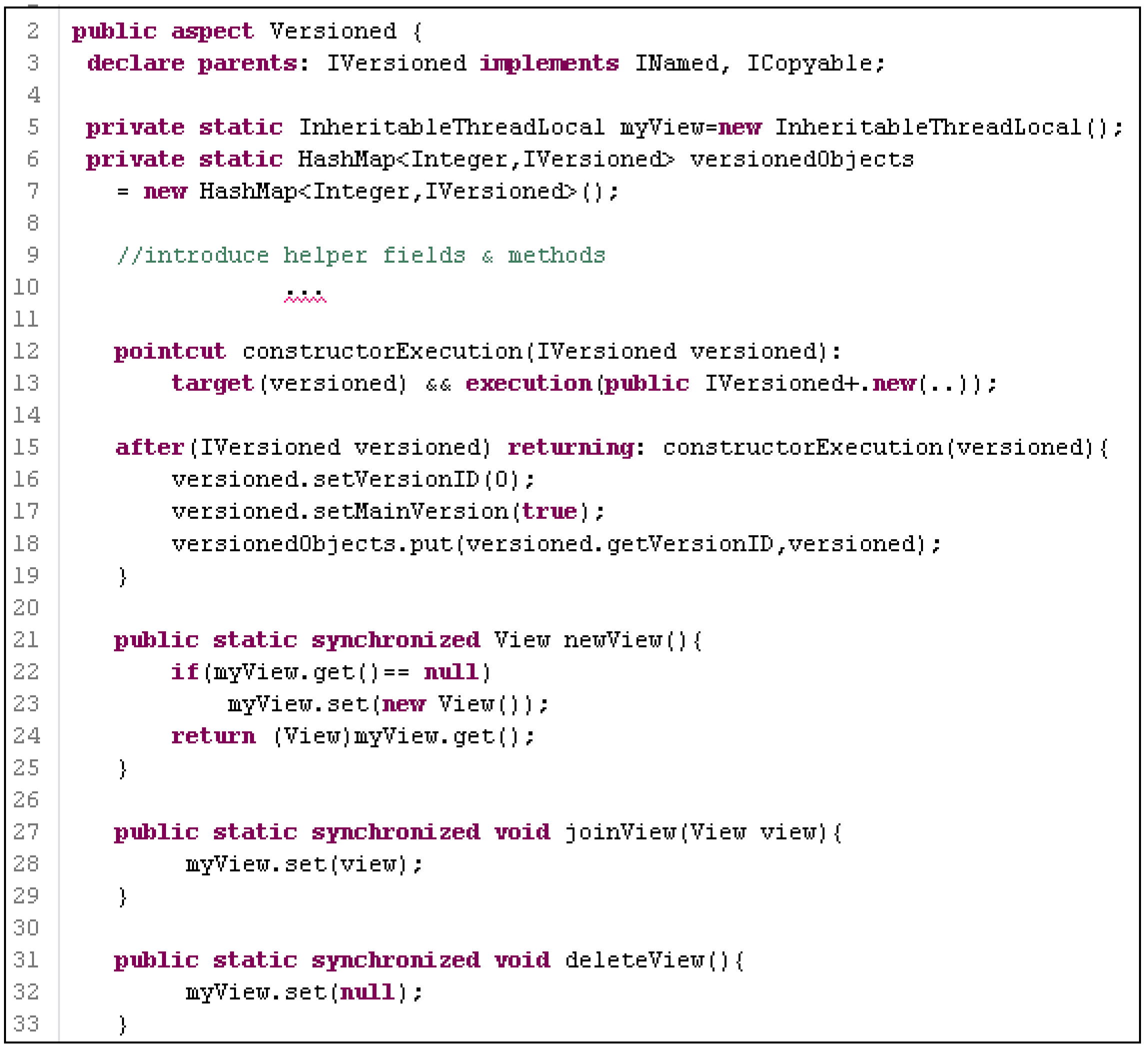
Summary of functionality

- *Versioned* encapsulates multiple versions of the state of an *IVersioned* object; with one version designated as the *main version*.
- *Versioned* objects are linked to the views of transactions (i.e., each transaction has its own perspective of a transactional object).
- A method invocation on a transactional object is directed either to the version in the view of the invoking transaction (if the calling transaction has joined a specific view) or the main version of the target object.
- *Versioned* also provides operations for managing transaction views and the versions of an object.

Implementation

Part I of the *Versioned* aspect implementation is presented in figure 5.12. The functionalities provided by the *Named* and *Copyable* aspects are requested on line 3. *Copyable* is required for duplicating the state of an object, *Named* is required for uniquely identifying an object.

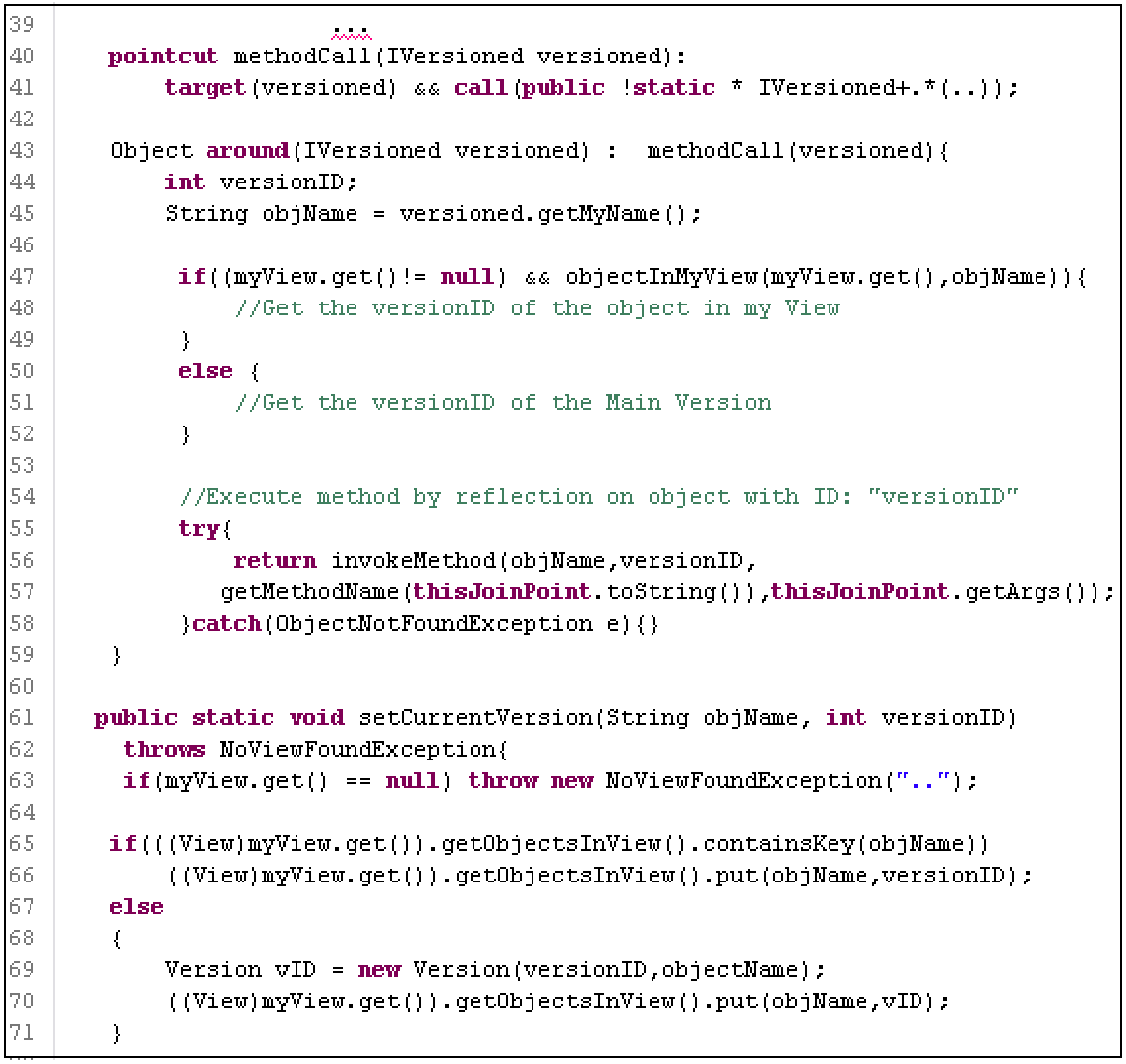
InheritableThreadLocal (line 5) is used for assigning views to threads (i.e., transactions). Lines 21 to 33 provides the methods for creating a new view (*View newView()*), joining an existing view (*joinView(View)*) and destroying the view of a thread (*deleteView()*). The constructor execution of a *Versioned* object is captured at line 12 and the version ID initialized with an *after* advice (lines 15 to 19). By default, objects created with the constructor are considered the *main version*.



```
public aspect Versioned {
 declare parents: IVersioned implements INamed, ICopyable;

 private static InheritableThreadLocal myView=new InheritableThreadLocal();
 private static HashMap<Integer,IVersioned> versionedObjects
   = new HashMap<Integer,IVersioned>();

   //introduce helper fields & methods
                ...

   pointcut constructorExecution(IVersioned versioned):
       target(versioned) && execution(public IVersioned+.new(..));

   after(IVersioned versioned) returning: constructorExecution(versioned){
       versioned.setVersionID(0);
       versioned.setMainVersion(true);
       versionedObjects.put(versioned.getVersionID,versioned);
   }

   public static synchronized View newView(){
       if(myView.get()== null)
           myView.set(new View());
       return (View)myView.get();
   }

   public static synchronized void joinView(View view){
        myView.set(view);
   }

   public static synchronized void deleteView(){
        myView.set(null);
   }
```

Figure 5.12: Implementation of Versioned – Part I

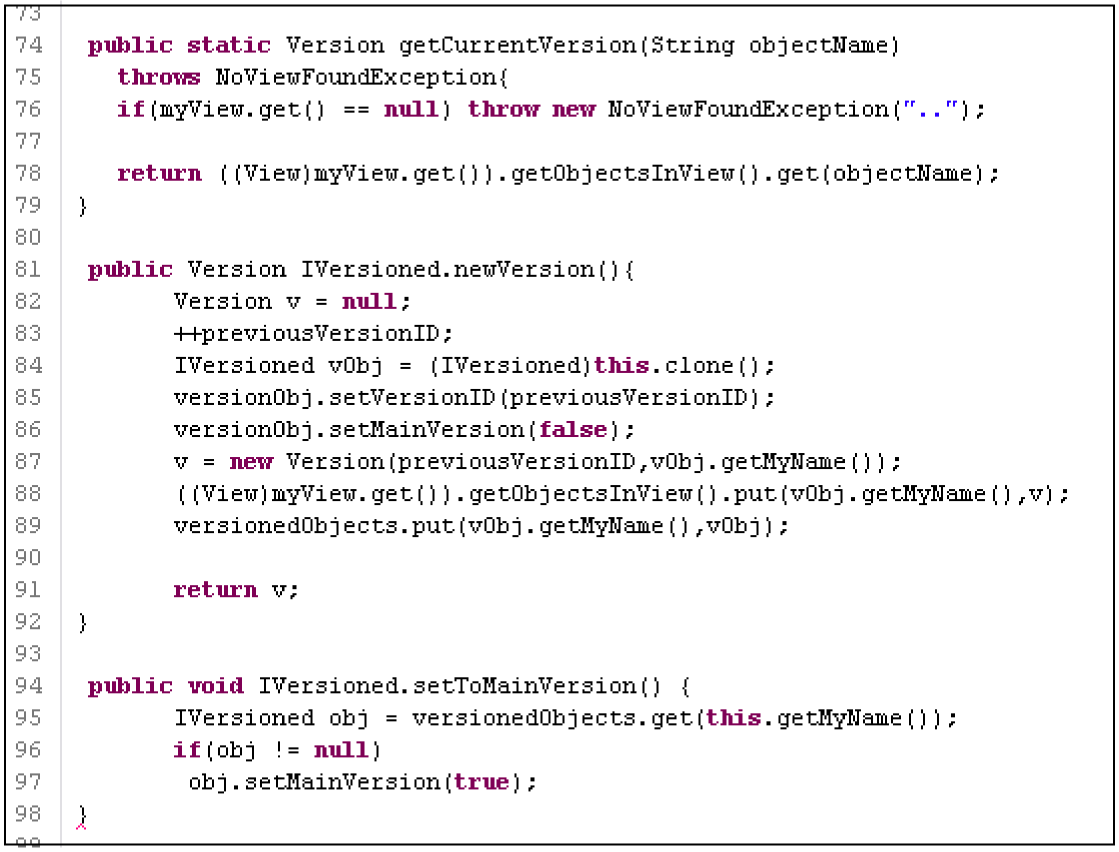
Part II of the *Versioned* aspect implementation is presented in figure 5.13. Besides being the *main version (MVObject)*, *MVObject* also serves as a handle to all the other versions of an object. The method call poincut (lines 40 to 41) captures calls to *MVObject* and the *around* advice (lines 43 to 59) is used for determining the appropriate version of the object on which to execute the operation. The version of the object in the *View* of the calling transaction is used, if one exist (lines 47 to 49); otherwise, the operation is executed on the *main version* (lines 50 to 52). Once determined, the framework executes the operation by reflection (lines 54 to 59) on the target version of the object. The use of the *around* advice is critical to the *Versioned* aspect implementation. This is because the captured join point always occurs on the *main version* but the target-version of the operation might be different; hence, the intercepted join point must be bypassed (by omitting the *proceed()* statement).



```
39                    ...
40    pointcut methodCall(IVersioned versioned):
41        target(versioned) && call(public !static * IVersioned+.*(..));
42
43    Object around(IVersioned versioned) :  methodCall(versioned){
44        int versionID;
45        String objName = versioned.getMyName();
46
47         if((myView.get()!= null) && objectInMyView(myView.get(),objName)){
48             //Get the versionID of the object in my View
49         }
50         else {
51             //Get the versionID of the Main Version
52         }
53
54         //Execute method by reflection on object with ID: "versionID"
55         try{
56             return invokeMethod(objName,versionID,
57            getMethodName(thisJoinPoint.toString()),thisJoinPoint.getArgs());
58         }catch(ObjectNotFoundException e){}
59    }
60
61   public static void setCurrentVersion(String objName, int versionID)
62     throws NoViewFoundException{
63     if(myView.get() == null) throw new NoViewFoundException("..");
64
65    if(((View)myView.get()).getObjectsInView().containsKey(objName))
66        ((View)myView.get()).getObjectsInView().put(objName,versionID);
67    else
68    {
69        Version vID = new Version(versionID,objectName);
70        ((View)myView.get()).getObjectsInView().put(objName,vID);
71    }
```

Figure 5.13: Implementation of Versioned – Part II

The code on lines 61 to 71 provides a method (*setCurrentVersion(...)*) for modifying the current version of an object associated with the view of a given thread.



```
74   public static Version getCurrentVersion(String objectName)
75     throws NoViewFoundException{
76     if(myView.get() == null) throw new NoViewFoundException("..");
77
78     return ((View)myView.get()).getObjectsInView().get(objectName);
79  }
80
81   public Version IVersioned.newVersion(){
82         Version v = null;
83         ++previousVersionID;
84         IVersioned vObj = (IVersioned)this.clone();
85         versionObj.setVersionID(previousVersionID);
86         versionObj.setMainVersion(false);
87         v = new Version(previousVersionID,vObj.getMyName());
88         ((View)myView.get()).getObjectsInView().put(vObj.getMyName(),v);
89         versionedObjects.put(vObj.getMyName(),vObj);
90
91         return v;
92  }
93
94   public void IVersioned.setToMainVersion() {
95         IVersioned obj = versionedObjects.get(this.getMyName());
96         if(obj != null)
97          obj.setMainVersion(true);
98  }
```

Figure 5.14: Implementation of Versioned – Part III

Part III (figure 5.14) of this implementation provides additional static crosscutting behaviours required by the *Versioned* aspect. The method for querying the current version of an object associated with the view a given thread is presented on lines 74 to 79 (*getCurrentVersion(...)*). The code on lines 81 to 98 provides methods for creating a new version of an object (*Version newVersion()*), and for designating a given version of an object as the main version (*setToMainVersion()*).

Discussion

There is the danger of infinite recursion in this implementation since *Versioned* has to intercept method invocations on *Versioned* objects (*the call phase*), and then execute the methods by reflection on the appropriate version (*the execution phase*). The inability of the call poincut (lines 40 to 41 – *figure 5.12*) in picking out reflective calls or executions has proven beneficial in this context. It picks out *the call phase* but not *the execution phase* of a method invocation, eluding infinite recursion. The execution pointcut (*execution(public * IVersioned+.*(..))*), if used on its own will result in infinite recursion since both *the call phase* and *the execution phase* of a method invocation will be captured. However, the net effect of composing the method execution join point with *cflow()* and *adviceexecution()* (producing *execution(public * IVersioned+.*(..)) && !cflow(adviceexecution())*) is equivalent to the call pointcut.

This implementation shares two of the three drawbacks discussed in section 5.3.4 (*Shared*), namely: *Weak aspect-to-class binding* and *Reflection/Super-class method execution dilemma*. The former, because the calls or executions of indirectly introduced methods on *Versioned* objects cannot be captured; hence, such methods will not be executed on the correct version of the object. The later, because reflective calls or execution of non-overridden inherited methods are also not intercepted, incurring similar ramifications as the former deficiency.

5.3.7 Tracked

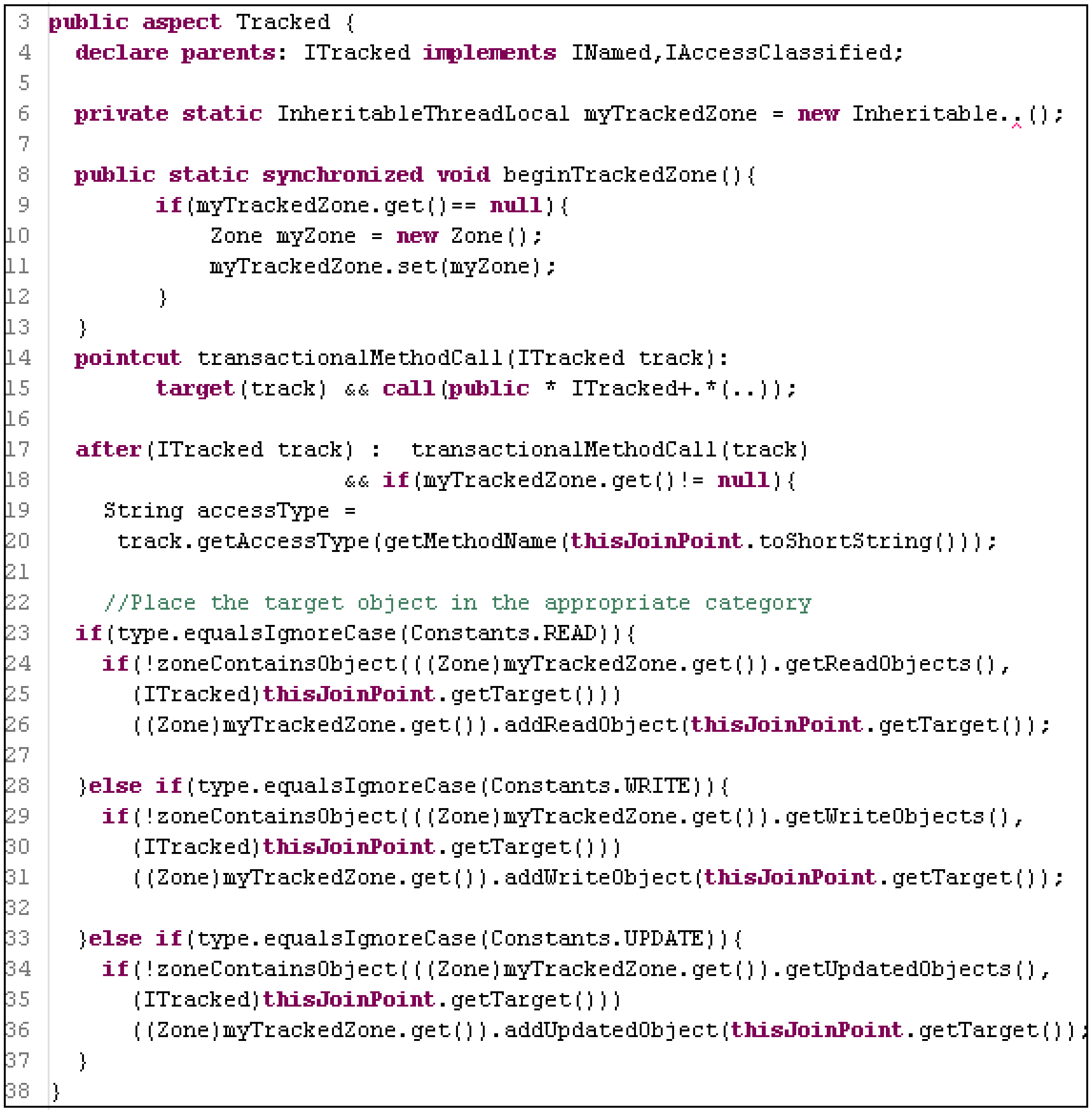
Summary of functionality

- Provides operations for defining a zone (in the execution of a thread) in which object accesses are monitored.
- Objects accessed within the defined zone are divided into three categories: *read*, *written-to* and *updated*.
- Provides operations for requesting the set of objects *read*, *written-to* or *updated* by a transaction.

Implementation

Figure 5.15 presents the first half of the implementation of the *Tracked* aspect. The *Tracked* aspect depends on *AccessClassified* (line 4) for distinguishing between *observer* and *modified* operations, and on *Named*, to avoid tracking different copies of the same transactional object. *InheritableThreadLocal* (line 6) is used for designating the track zone of a thread. Track zones are requested by executing the aspect method *beginTrackZone()* - lines 8 to 13.

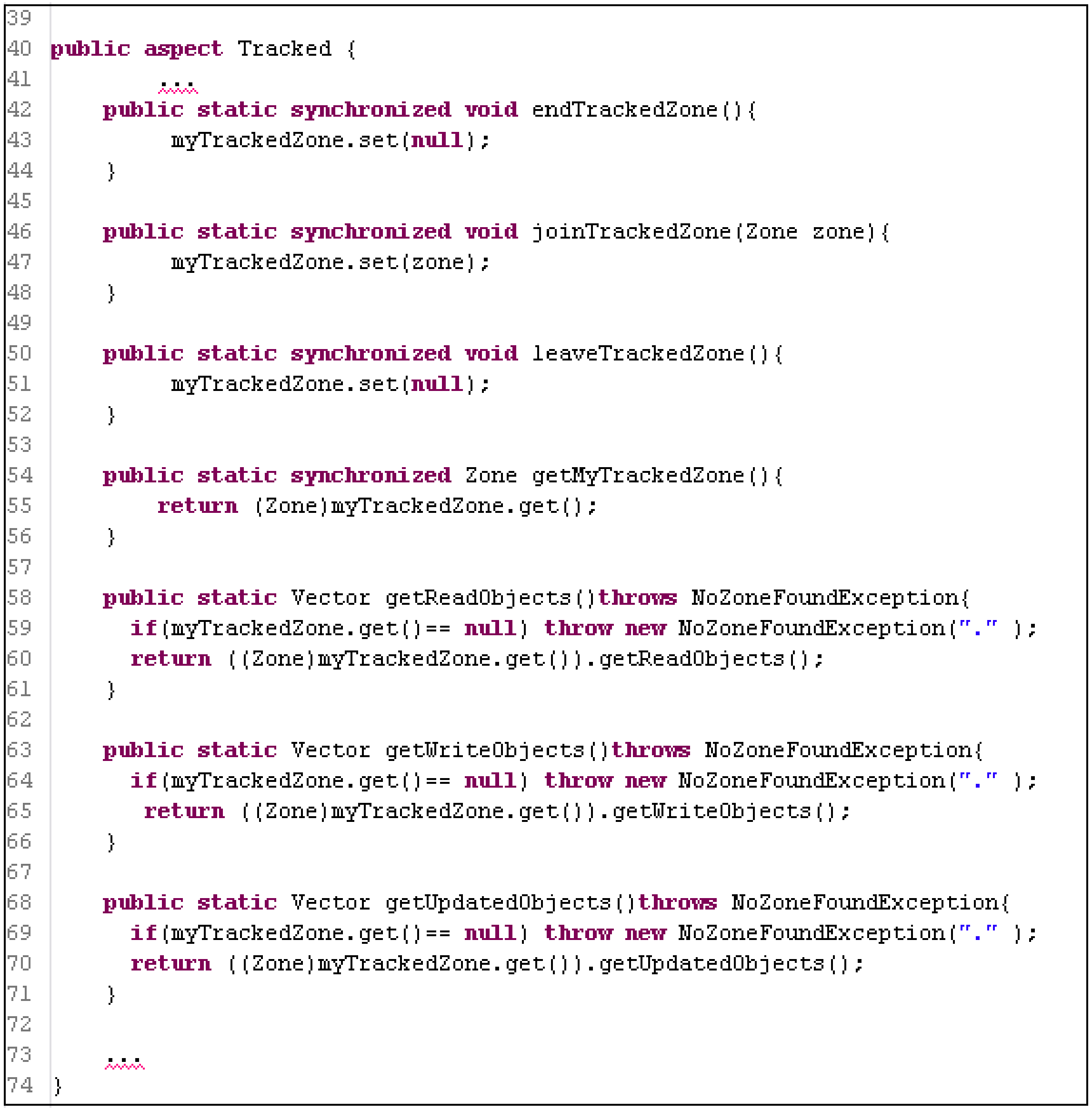
The poincut for capturing accesses to *Tracked* objects is defined on lines 14 to 15. The *after* advice (lines 17 to 38) places the target object in the appropriate category with the assistance received from the *AccessClassified* aspect (lines 19 to 20). This categorization is performed only for threads with an existing track zone (line 18).



```
public aspect Tracked {
  declare parents: ITracked implements INamed,IAccessClassified;

  private static InheritableThreadLocal myTrackedZone = new Inheritable.();

  public static synchronized void beginTrackedZone(){
        if(myTrackedZone.get()== null){
            Zone myZone = new Zone();
            myTrackedZone.set(myZone);
        }
  }
  pointcut transactionalMethodCall(ITracked track):
        target(track) && call(public * ITracked+.*(..));

  after(ITracked track) :  transactionalMethodCall(track)
                     && if(myTrackedZone.get()!= null){
    String accessType =
     track.getAccessType(getMethodName(thisJoinPoint.toShortString()));

    //Place the target object in the appropriate category
  if(type.equalsIgnoreCase(Constants.READ)){
    if(!zoneContainsObject(((Zone)myTrackedZone.get()).getReadObjects(),
      (ITracked)thisJoinPoint.getTarget()))
      ((Zone)myTrackedZone.get()).addReadObject(thisJoinPoint.getTarget());

  }else if(type.equalsIgnoreCase(Constants.WRITE)){
    if(!zoneContainsObject(((Zone)myTrackedZone.get()).getWriteObjects(),
      (ITracked)thisJoinPoint.getTarget()))
      ((Zone)myTrackedZone.get()).addWriteObject(thisJoinPoint.getTarget());

  }else if(type.equalsIgnoreCase(Constants.UPDATE)){
    if(!zoneContainsObject(((Zone)myTrackedZone.get()).getUpdatedObjects(),
      (ITracked)thisJoinPoint.getTarget()))
      ((Zone)myTrackedZone.get()).addUpdatedObject(thisJoinPoint.getTarget());
  }
}
```

Figure 5.15: Implementation of Tracked – Part I

The second half of the implementation of the *Tracked* aspect is presented in figure 5.16. The code on lines 58 to 71 provides operations for requesting the set of objects *read* (*getReadObjects()*) *written-to* (*getWriteObjects()*) or *updated* (*getUpdatedObjects()*) by a transaction. The operations for managing track zones are presented on lines 42 to 56. A thread can end its track zone (*endTrackedZone()*), join an existing track zone (*joinTrackedZone()*), leave a track zone (*leaveTrackedZone()*), and request a reference to its track zone (*getMyTrackedZone()*).



```java

public aspect Tracked {
        ...
    public static synchronized void endTrackedZone(){
         myTrackedZone.set(null);
    }

    public static synchronized void joinTrackedZone(Zone zone){
         myTrackedZone.set(zone);
    }

    public static synchronized void leaveTrackedZone(){
         myTrackedZone.set(null);
    }

    public static synchronized Zone getMyTrackedZone(){
        return (Zone)myTrackedZone.get();
    }

    public static Vector getReadObjects()throws NoZoneFoundException{
      if(myTrackedZone.get()== null) throw new NoZoneFoundException("." );
      return ((Zone)myTrackedZone.get()).getReadObjects();
    }

    public static Vector getWriteObjects()throws NoZoneFoundException{
      if(myTrackedZone.get()== null) throw new NoZoneFoundException("." );
       return ((Zone)myTrackedZone.get()).getWriteObjects();
    }

    public static Vector getUpdatedObjects()throws NoZoneFoundException{
      if(myTrackedZone.get()== null) throw new NoZoneFoundException("." );
      return ((Zone)myTrackedZone.get()).getUpdatedObjects();
    }

    ...
}
```

Figure 5.16: Implementation of Tracked – Part II

Discussion

This implementation shares two of the three drawbacks discussed in section 5.3.4 (*Shared*), namely: *Weak aspect-to-class binding* (because the calls or executions of indirectly introduced methods on *Tracked* objects cannot be tracked) and *Reflection/Super-class method execution* *dilemma* (because reflective calls or execution of non-overridden inherited methods cannot be tracked). The ramifications of these deficiencies may be costly; for example, state changes executed on transactional objects by indirectly introduced or non-overridden inherited methods cannot be undone in the event of a transaction abort.

5.3.8 Recoverable

Summary of functionality

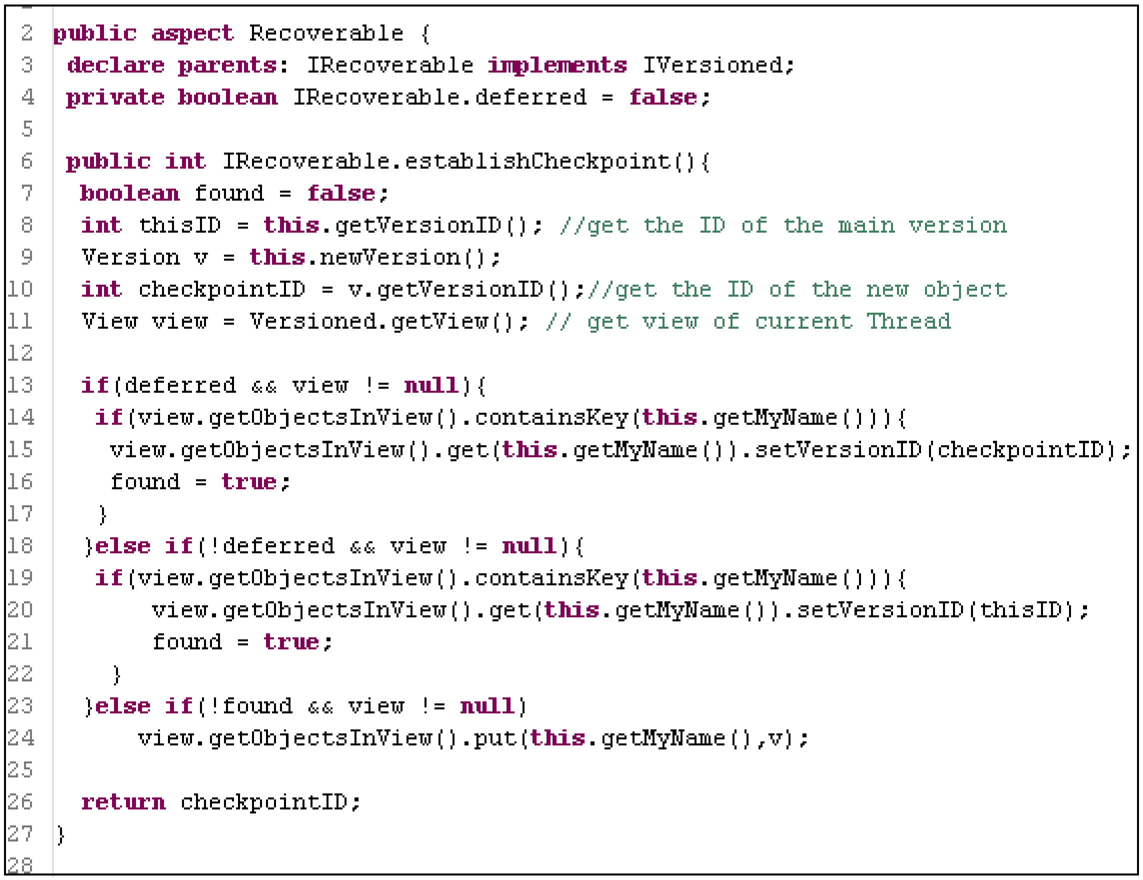
- An *IRecoverable* object knows how to take a snapshot of its state (i.e., establish a checkpoint), restore its state from a previously established checkpoint, discard a checkpoint and switch its update strategy to either *in-place* or *deferred* update.

Implementation

Figure 5.17 presents the first part of the implementation of the *Recoverable* aspect. The functionality provided by the *Versioned* aspect is requested on line 3. *Recoverable* relies on *Versioned* for creating a new version of an object (line 9). Line 4 introduces a variable for assigning the update strategy of an object. Lines 6 to 33 of this code introduces a method (*int establishCheckpoint()*) for establishing a checkpoint of an object. The return value of this method provides a handle to the established checkpoint.

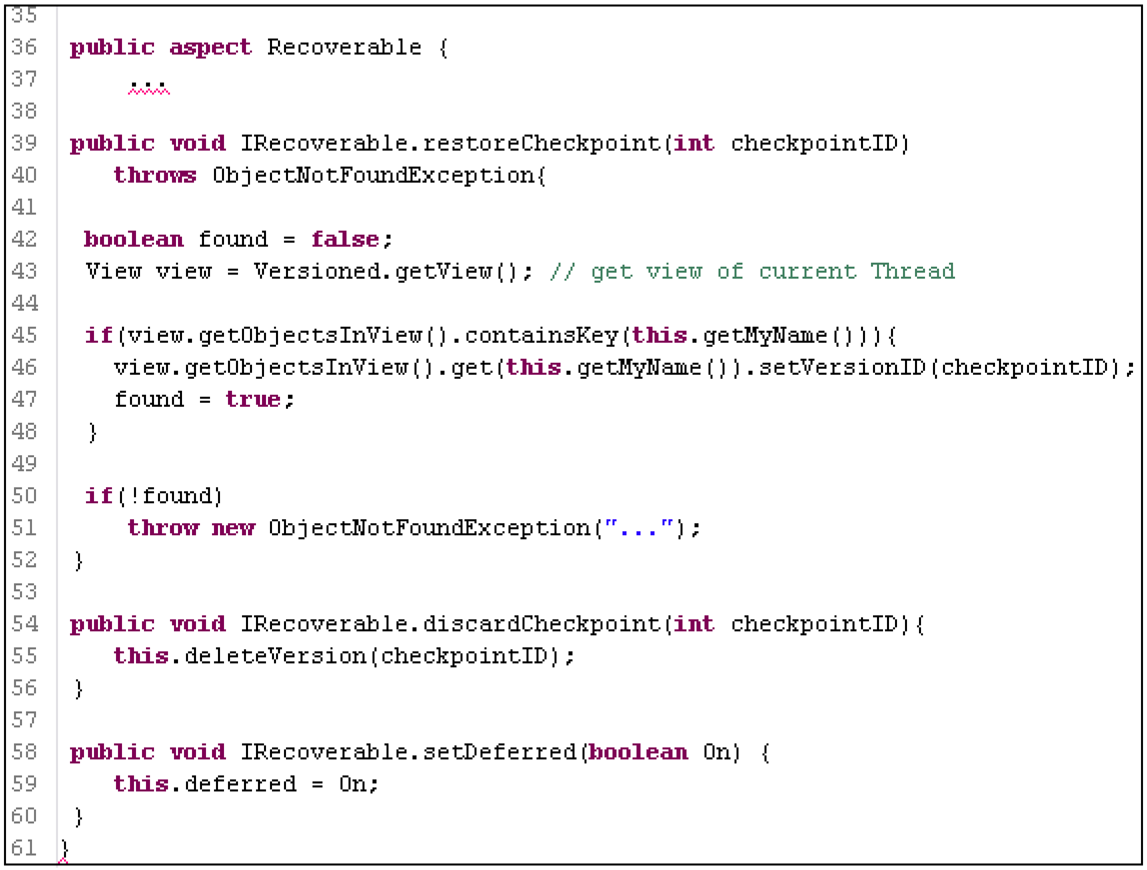
After creating a new version of an object (line 9), *Recoverable* has to determine whether this new version should be assigned to the *View* of the invoking transaction or not based on the current update strategy. If the current update strategy is *deferred* (lines 13 to 17), then the version of the object visible to the invoking transaction (i.e., in its *View*) is changed to the version of the newly created object (line

15). This results in a unique local copy of an object per transaction for the deferred update strategy (see section 3.2.3). If the current update strategy is *in-place* (lines 18 to 22), then the version of the object visible to the invoking transaction (i.e., in its *View*) continues to be the main version (line 20).



```
 2 public aspect Recoverable {
 3  declare parents: IRecoverable implements IVersioned;
 4  private boolean IRecoverable.deferred = false;
 5
 6  public int IRecoverable.establishCheckpoint(){
 7   boolean found = false;
 8   int thisID = this.getVersionID(); //get the ID of the main version
 9   Version v = this.newVersion();
10   int checkpointID = v.getVersionID();//get the ID of the new object
11   View view = Versioned.getView(); // get view of current Thread
12
13   if(deferred && view != null){
14    if(view.getObjectsInView().containsKey(this.getMyName())){
15     view.getObjectsInView().get(this.getMyName()).setVersionID(checkpointID);
16     found = true;
17    }
18   }else if(!deferred && view != null){
19    if(view.getObjectsInView().containsKey(this.getMyName())){
20        view.getObjectsInView().get(this.getMyName()).setVersionID(thisID);
21        found = true;
22     }
23   }else if(!found && view != null)
24       view.getObjectsInView().put(this.getMyName(),v);
25
26   return checkpointID;
27 }
28
```

Figure 5.17: Implementation of Recoverable – Part I



```
35
36  public aspect Recoverable {
37      ...
38
39  public void IRecoverable.restoreCheckpoint(int checkpointID)
40     throws ObjectNotFoundException{
41
42   boolean found = false;
43   View view = Versioned.getView(); // get view of current Thread
44
45   if(view.getObjectsInView().containsKey(this.getMyName())){
46     view.getObjectsInView().get(this.getMyName()).setVersionID(checkpointID);
47     found = true;
48   }
49
50   if(!found)
51      throw new ObjectNotFoundException("...");
52  }
53
54  public void IRecoverable.discardCheckpoint(int checkpointID){
55     this.deleteVersion(checkpointID);
56  }
57
58  public void IRecoverable.setDeferred(boolean On) {
59     this.deferred = On;
60  }
61 }
```

Figure 5.18: Implementation of Recoverable – Part II

Figure 5.18 presents the second half of the implementation of the *Recoverable* aspect. Lines 39 to 52 introduces the method (*restoreCheckpoint(CheckpointID)*) for restoring an object's state to a previously established checkpoint. The input parameter of this method is the checkpoint ID of a previously establish checkpoint. Restoring a checkpoint simply involves changing the version of the object in the *View* of the invoking transaction to the checkpoint ID of a previously establish checkpoint (line 46). The method for discarding a checkpoint is introduced on lines 54 to 56 (*discardCheckpoint(checkpointID)*), and lines 58 to 60 introduces the method for switching between update strategies (*setDeferred(Boolean)*).

Discussion

The *Recoverable* aspect provides a versatile implementation for undoing state changes on a transactional object. It supports multiple checkpoints of the state of an object and multiple rollbacks to any of the previous established checkpoints.

5.3.9 AutoRecoverable

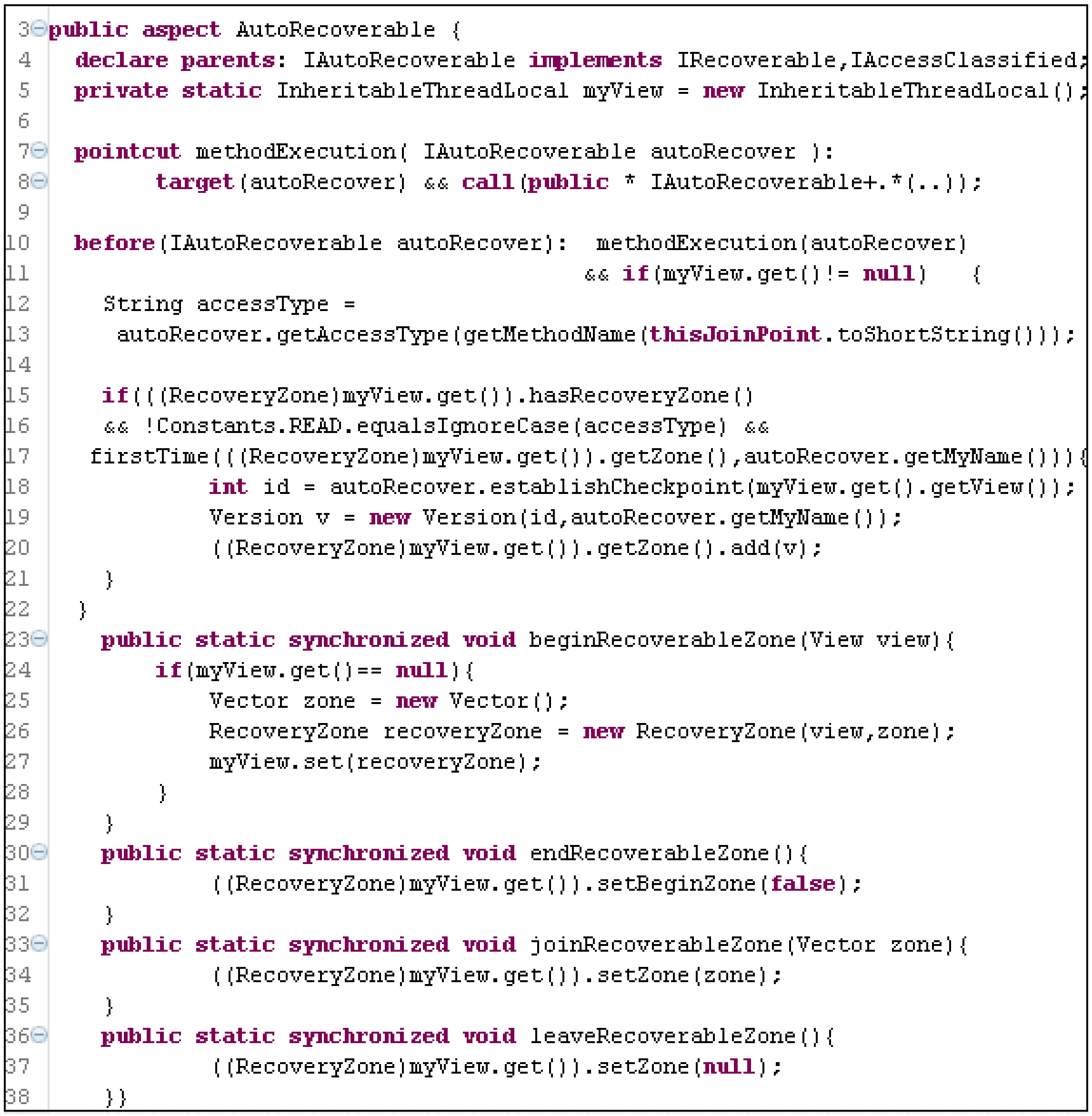
Summary of functionality

- This aspect provides operations for delimiting and managing recoverable zones (in the execution of a thread) in which object accesses are monitored.
- A checkpoint is automatically established the first time an *IAutoRecoverable* object is modified within the defined zone.

Implementation

The implementation of the *AutoRecoverable* aspect is presented in figure 5.19. This aspect relies on *AccessClassified* in distinguishing *modifier* operations from *observer* operations (line 4), and on *Recoverable* in establishing and rolling back checkpoints (line 4). *InheritableThreadLocal* (line 5) is once more used for designating the recoverable zone of a thread. Recoverable zones are requested by executing the aspect method *beginRecoverableZone(View)* - lines 23 to 29.

The poincut for intercepting operation calls to *AutoRecoverable* objects is presented on lines 7 to 8. The *before* advice (lines 10 to 21) is executed only if the current executing thread had previously defined a recoverable zone (line 11). The access type of the operation is determined (lines 12 - 13), and a checkpoint established for the target object if the operation is a *modifier* (lines 15 - 21) and a checkpoint has not been previously established for this object (line 17).



```
3  public aspect AutoRecoverable {
4    declare parents: IAutoRecoverable implements IRecoverable,IAccessClassified;
5    private static InheritableThreadLocal myView = new InheritableThreadLocal();
6
7    pointcut methodExecution( IAutoRecoverable autoRecover ):
8          target(autoRecover) && call(public * IAutoRecoverable+.*(..));
9
10   before(IAutoRecoverable autoRecover):  methodExecution(autoRecover)
11                                         && if(myView.get()!= null)   {
12     String accessType =
13      autoRecover.getAccessType(getMethodName(thisJoinPoint.toShortString()));
14
15     if(((RecoveryZone)myView.get()).hasRecoveryZone()
16     && !Constants.READ.equalsIgnoreCase(accessType) &&
17    firstTime(((RecoveryZone)myView.get()).getZone(),autoRecover.getMyName())){
18           int id = autoRecover.establishCheckpoint(myView.get().getView());
19           Version v = new Version(id,autoRecover.getMyName());
20           ((RecoveryZone)myView.get()).getZone().add(v);
21     }
22   }
23     public static synchronized void beginRecoverableZone(View view){
24         if(myView.get()== null){
25             Vector zone = new Vector();
26             RecoveryZone recoveryZone = new RecoveryZone(view,zone);
27             myView.set(recoveryZone);
28         }
29     }
30     public static synchronized void endRecoverableZone(){
31             ((RecoveryZone)myView.get()).setBeginZone(false);
32     }
33     public static synchronized void joinRecoverableZone(Vector zone){
34             ((RecoveryZone)myView.get()).setZone(zone);
35     }
36     public static synchronized void leaveRecoverableZone(){
37             ((RecoveryZone)myView.get()).setZone(null);
38     }}
```

Figure 5.19: Implementation of AutoRecoverable

The methods for managing recoverable zones are presented on lines 30 to 38 of the implementation. A thread can end its recoverable zone (*endRecoverableZone()*), join an existing recoverable zone (*joinRecoverableZone()*), or leave a previously joined recoverable zone (*leaveRecoverableZone()*).

Discussion

This implementation also shares two of the three drawbacks discussed in section 5.3.4 (*Shared*): *Weak aspect-to-class binding* (because the calls or executions of indirectly introduced methods on an *IAutoRecoverable* object cannot be captured and check-pointed) and *Reflection/Super-class method execution dilemma* (because reflective calls or execution of non-overridden inherited methods cannot be not captured). The ramifications of these deficiencies may also be costly; for instance, state changes executed on transactional objects by indirectly introduced or non-overridden inherited methods cannot be *rolled-back* by *Recoverable* in the event of a transaction abort.

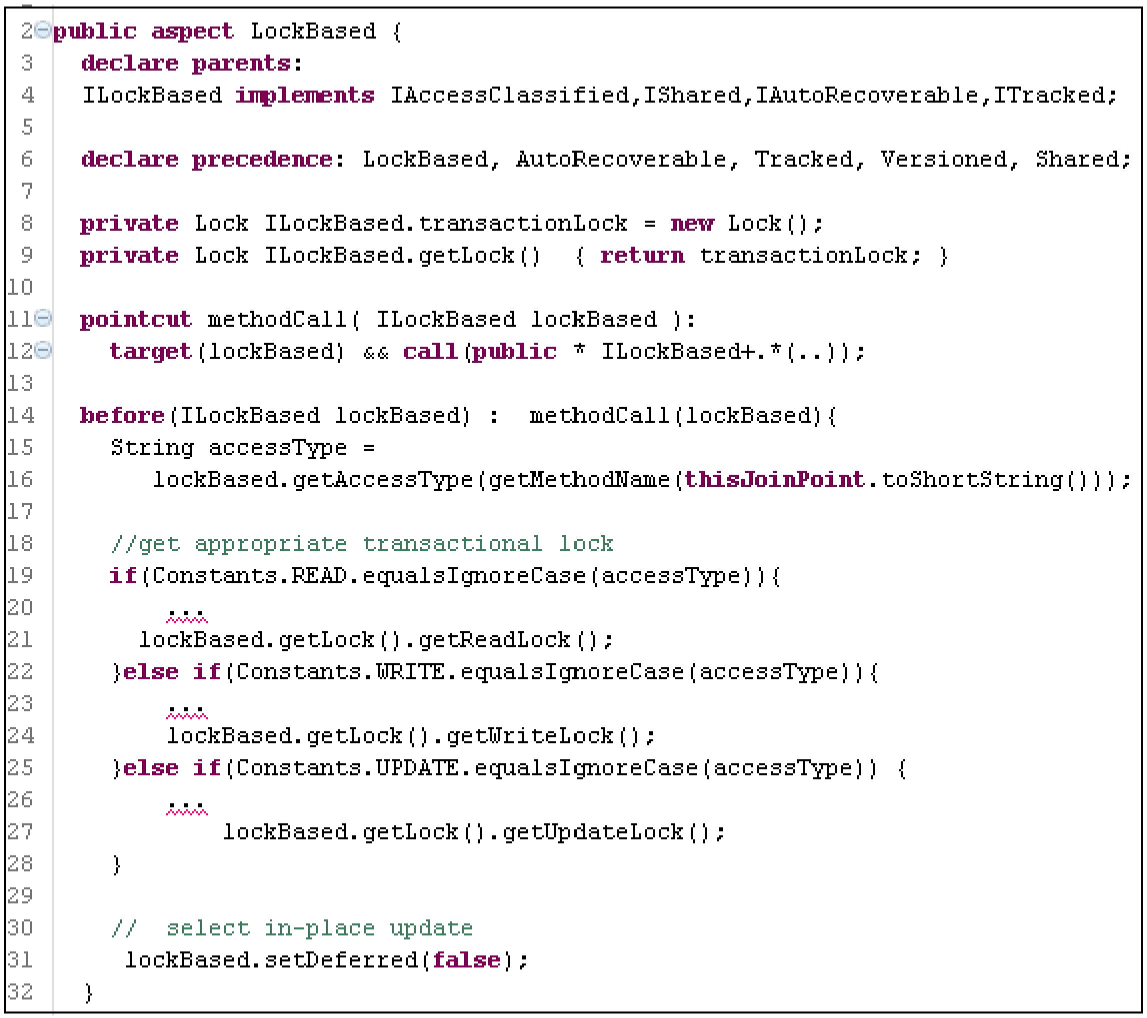
5.3.10 LockBased

Summary of functionality

- This aspect provides support for *pessimistic* lock-based concurrency control with *in-place* update.
- The appropriate transactional lock (*Read*, *Write* or *Update*) must be acquired before invoking an operation on a *LockBased* object.
- Unlike *Shared*, transactional locks are released only after the outcome of the transaction is known.

Implementation

Figure 5.20 presents the first part of the implementation of the *LockBased* aspect. *LockBased* (lines 3 to 4) relies on *AccessClassified* (in distinguishing between *Read*, *Write* and *Update* operations), *Shared* (in preventing threads within a transaction from currently modify an object's state), *AutoRecoverable* (in facilitating the undo functionality - in event of a transaction abort) and *Tracked* (in order to remember to release the previously acquired transactional locks upon a transaction commit or abort).



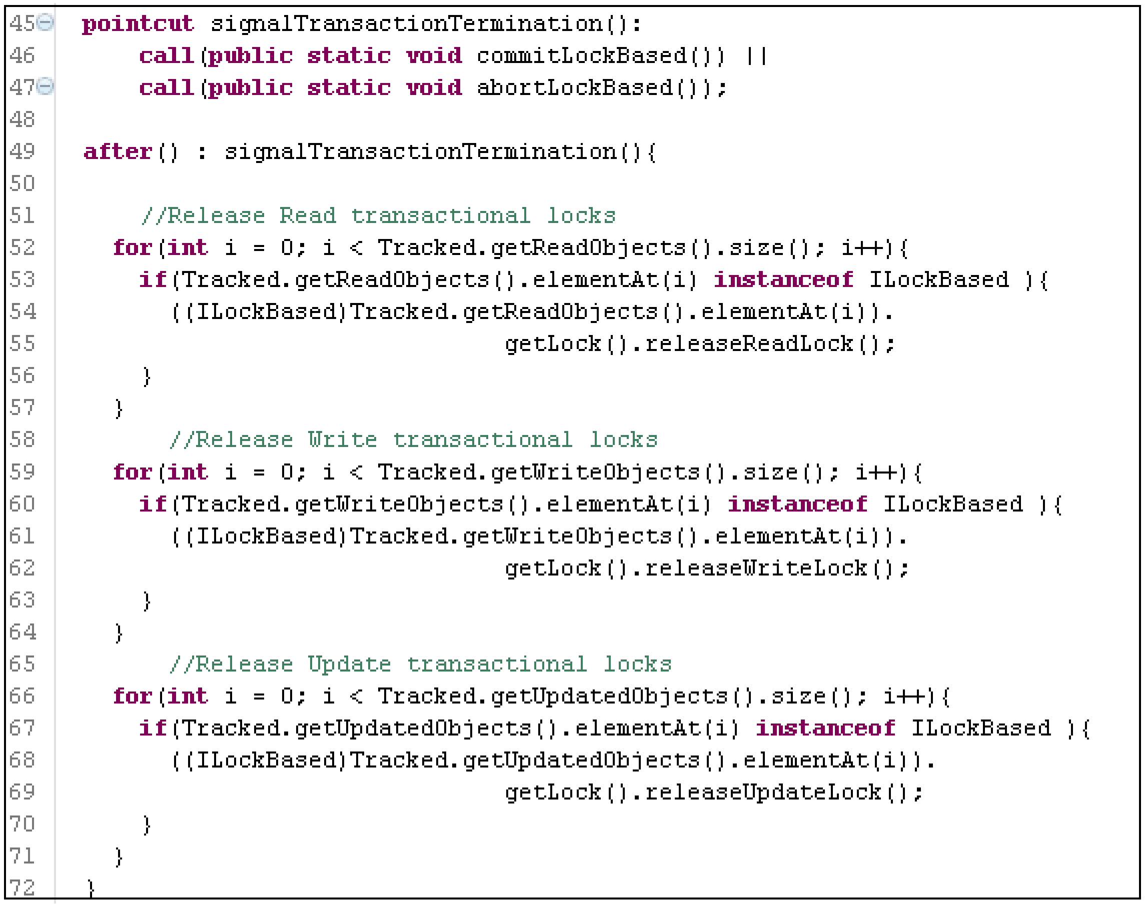
```
2  public aspect LockBased {
3    declare parents:
4    ILockBased implements IAccessClassified,IShared,IAutoRecoverable,ITracked;
5
6    declare precedence: LockBased, AutoRecoverable, Tracked, Versioned, Shared;
7
8    private Lock ILockBased.transactionLock = new Lock();
9    private Lock ILockBased.getLock()  { return transactionLock; }
10
11   pointcut methodCall( ILockBased lockBased ):
12     target(lockBased) && call(public * ILockBased+.*(..));
13
14   before(ILockBased lockBased) :  methodCall(lockBased){
15     String accessType =
16        lockBased.getAccessType(getMethodName(thisJoinPoint.toShortString()));
17
18     //get appropriate transactional lock
19     if(Constants.READ.equalsIgnoreCase(accessType)){
20         ...
21       lockBased.getLock().getReadLock();
22     }else if(Constants.WRITE.equalsIgnoreCase(accessType)){
23         ...
24         lockBased.getLock().getWriteLock();
25     }else if(Constants.UPDATE.equalsIgnoreCase(accessType)) {
26         ...
27             lockBased.getLock().getUpdateLock();
28     }
29
30     //  select in-place update
31      lockBased.setDeferred(false);
32   }
```

Figure 5.20: Implementation of LockBased aspect – Part I

The execution order of these aspects is crucial. An unspecified precedence may not produce the desired result and has the risk of becoming deadlocked. For instance, one transaction might have the transactional lock of an object and need the mutual exclusion *Shared* lock whereas another transaction has the mutual exclusion *Shared* lock of the same object and is waiting for the transactional lock. The desired execution order in this case is: *LockBased*, *AutoRecoverable, Tracked, Versioned* and *Shared* (line 6) because the update strategy has to be set to *in-place* by *LockBased* before *AutoRecoverable* executes, the object is then *Tracked*, the operation directed to the *main version* by

Versioned, and mutual exclusion to the state of the object ensured by *Shared*.

The field and method necessary to support transactional locks are introduced on lines 8 to 9. The pointcut for intercepting calls to *LockBased* objects is presented on lines 11 to 12. The *before* advice (lines 14 to 32) forces a transaction to obtain the appropriate transactional lock on a *LockBased* object, sets the update strategy to *in-place* (line 31) before permitting the execution of an operation. *AccessClassified* is used in determining the type of transactional lock associated with a given operation (lines 15 to 16).



```
pointcut signalTransactionTermination():
    call(public static void commitLockBased()) ||
    call(public static void abortLockBased());

after() : signalTransactionTermination(){

    //Release Read transactional locks
  for(int i = 0; i < Tracked.getReadObjects().size(); i++){
    if(Tracked.getReadObjects().elementAt(i) instanceof ILockBased ){
      ((ILockBased)Tracked.getReadObjects().elementAt(i)).
                              getLock().releaseReadLock();
    }
  }
      //Release Write transactional locks
  for(int i = 0; i < Tracked.getWriteObjects().size(); i++){
    if(Tracked.getWriteObjects().elementAt(i) instanceof ILockBased ){
      ((ILockBased)Tracked.getWriteObjects().elementAt(i)).
                              getLock().releaseWriteLock();
    }
  }
      //Release Update transactional locks
  for(int i = 0; i < Tracked.getUpdatedObjects().size(); i++){
    if(Tracked.getUpdatedObjects().elementAt(i) instanceof ILockBased ){
      ((ILockBased)Tracked.getUpdatedObjects().elementAt(i)).
                              getLock().releaseUpdateLock();
    }
  }
}
```

Figure 5.21: Implementation of LockBased aspect – Part II

The second part of the *LockBased* aspect implementation shows how previously acquired transactional locks are released (figure 5.21). Unlike *Shared*, *LockBased* holds onto the transactional locks until the outcome of the transaction is known. One of two global methods (*commitLockBased()* or *abortLockBased()*) can be used in signalling the outcome of a transaction. The pointcut for intercepting calls to these methods is presented on lines 45 to 47. The *after* advice (lines 49 to 72) then releases the acquired transactional locks with the assistance of *Tracked*. *Tracked* provides *LockBased* with the set of objects *Read* (line 52), *Written-to* (line 59), or *updated* (line 66). These objects are down-casted to *ILockBased* (lines 54, 61 and 68) before releasing the previously acquired locks.

Discussion

The *LockBased* aspect also has two of the three drawbacks discussed in section 5.3.4 (*Shared*), namely: *Weak aspect-to-class binding* (because the calls or executions of indirectly introduced methods on an *ILockBased* object cannot be captured; consequently, cannot be forced to obtain transactional locks) and *Reflection/Super-class method execution dilemma* (because reflective calls or executions of non-overridden inherited methods cannot also be forced to obtain transactional locks). As a result, the state consistency of *ILockBased* objects cannot be absolutely guaranteed.

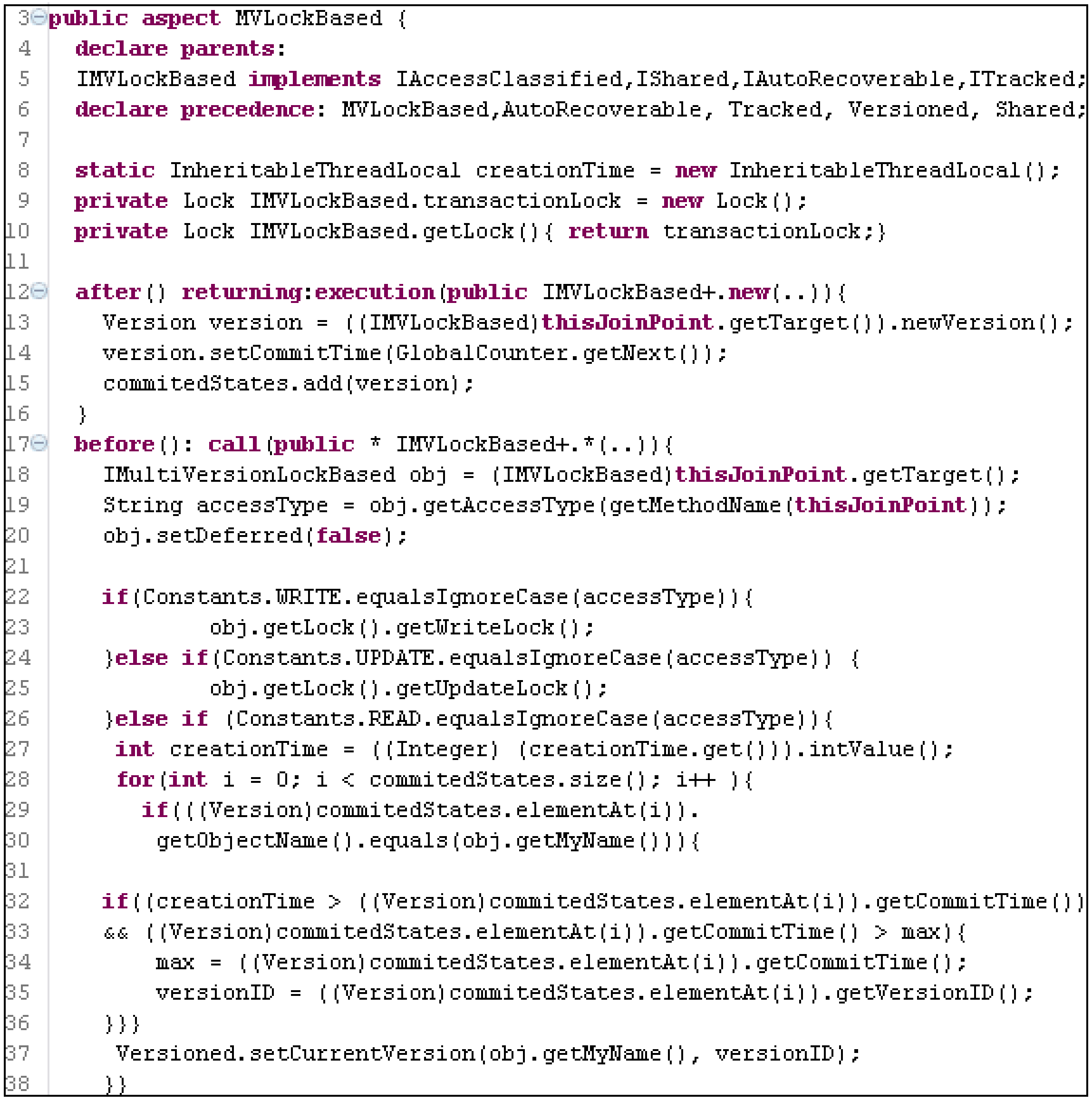
5.3.11 Multi-Version LockBased

Summary of functionality

- This aspect provides support for *pessimistic* Multi-Version Lock-Based concurrency control with *in-place* update.
- It encapsulates two sets of states per transactional object: a *history of committed states* (HCS) and the *main version*.

- *Write* and *Update* operations are executed on the *main version* and must acquire the appropriate transactional lock before invoking an operation on the object.
- *Read* operations are executed on the appropriate objects in the HCS; therefore, they do not have to acquire either *Shared* locks or transactional locks.

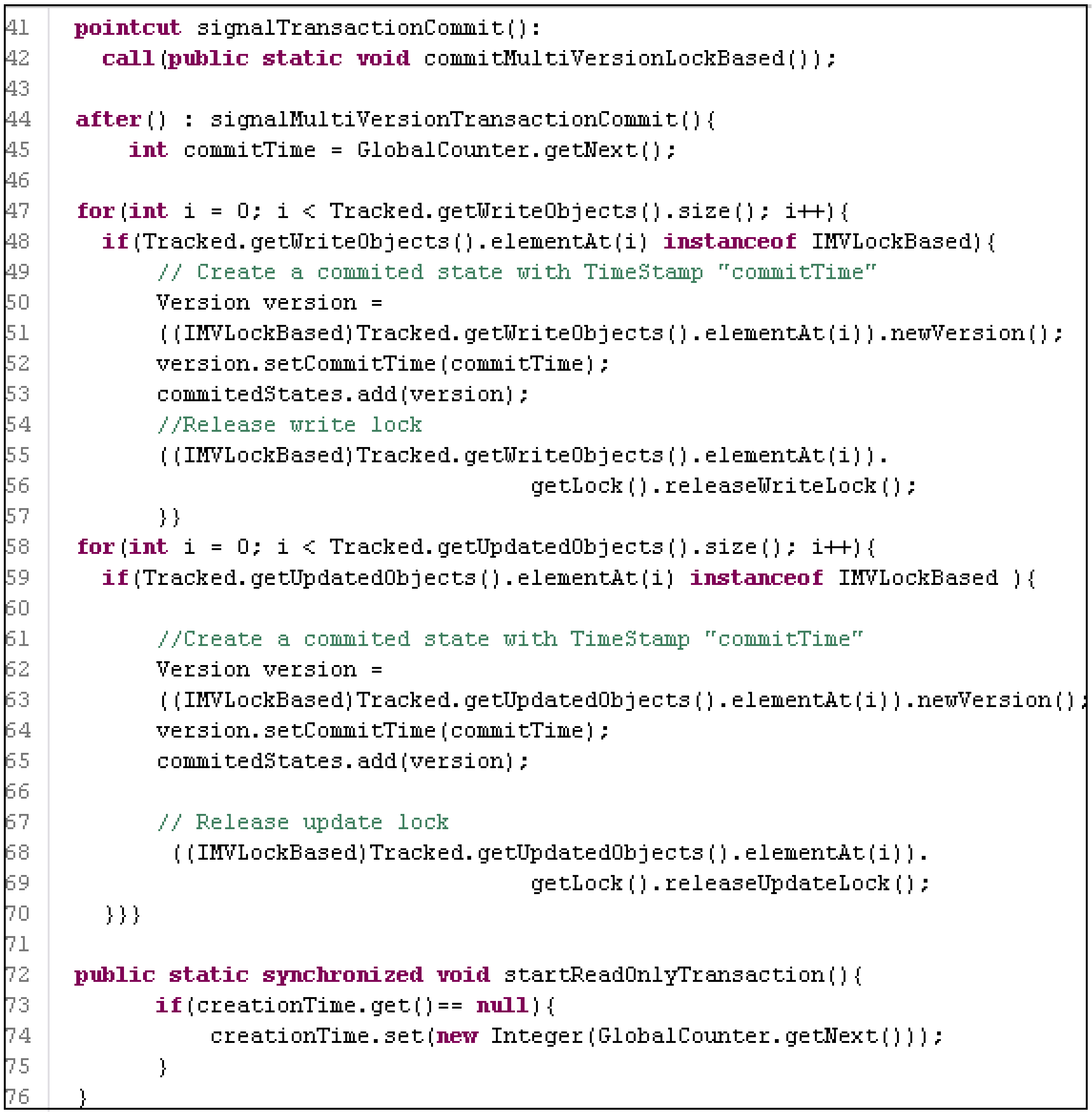
Implementation



```
public aspect MVLockBased {
  declare parents:
  IMVLockBased implements IAccessClassified,IShared,IAutoRecoverable,ITracked;
  declare precedence: MVLockBased,AutoRecoverable, Tracked, Versioned, Shared;

  static InheritableThreadLocal creationTime = new InheritableThreadLocal();
  private Lock IMVLockBased.transactionLock = new Lock();
  private Lock IMVLockBased.getLock(){ return transactionLock;}

  after() returning:execution(public IMVLockBased+.new(..)){
    Version version = ((IMVLockBased)thisJoinPoint.getTarget()).newVersion();
    version.setCommitTime(GlobalCounter.getNext());
    commitedStates.add(version);
  }
  before(): call(public * IMVLockBased+.*(..)){
    IMultiVersionLockBased obj = (IMVLockBased)thisJoinPoint.getTarget();
    String accessType = obj.getAccessType(getMethodName(thisJoinPoint));
    obj.setDeferred(false);

    if(Constants.WRITE.equalsIgnoreCase(accessType)){
            obj.getLock().getWriteLock();
    }else if(Constants.UPDATE.equalsIgnoreCase(accessType)) {
            obj.getLock().getUpdateLock();
    }else if (Constants.READ.equalsIgnoreCase(accessType)){
     int creationTime = ((Integer) (creationTime.get())).intValue();
     for(int i = 0; i < commitedStates.size(); i++ ){
       if(((Version)commitedStates.elementAt(i)).
        getObjectName().equals(obj.getMyName())){

    if((creationTime > ((Version)commitedStates.elementAt(i)).getCommitTime())
    && ((Version)commitedStates.elementAt(i)).getCommitTime() > max){
        max = ((Version)commitedStates.elementAt(i)).getCommitTime();
        versionID = ((Version)commitedStates.elementAt(i)).getVersionID();
    }}}
     Versioned.setCurrentVersion(obj.getMyName(), versionID);
    }}
```

Figure 5.22: Implementation of Multi-Version LockBased – Part I

Figure 5.22 presents the first half of the implementation of this aspect. The *declare parents* (lines 4 to 5) and *declare precedence* (line 6) statements of this implementation are the same as in the *LockBased* aspect for similar reasons. *Write* and *Update* operations are also treated similarly – they must acquire the associated transactional locks on the *main version* and set the update strategy to *in-place* before executing (lines 21-25, 19 respectively).



```
pointcut signalTransactionCommit():
  call(public static void commitMultiVersionLockBased());

after() : signalMultiVersionTransactionCommit(){
    int commitTime = GlobalCounter.getNext();

for(int i = 0; i < Tracked.getWriteObjects().size(); i++){
  if(Tracked.getWriteObjects().elementAt(i) instanceof IMVLockBased){
      // Create a commited state with TimeStamp "commitTime"
      Version version =
      ((IMVLockBased)Tracked.getWriteObjects().elementAt(i)).newVersion();
      version.setCommitTime(commitTime);
      commitedStates.add(version);
      //Release write lock
      ((IMVLockBased)Tracked.getWriteObjects().elementAt(i)).
                                  getLock().releaseWriteLock();
      }}
for(int i = 0; i < Tracked.getUpdatedObjects().size(); i++){
  if(Tracked.getUpdatedObjects().elementAt(i) instanceof IMVLockBased ){

      //Create a commited state with TimeStamp "commitTime"
      Version version =
      ((IMVLockBased)Tracked.getUpdatedObjects().elementAt(i)).newVersion();
      version.setCommitTime(commitTime);
      commitedStates.add(version);

      // Release update lock
       ((IMVLockBased)Tracked.getUpdatedObjects().elementAt(i)).
                                  getLock().releaseUpdateLock();
  }}}

public static synchronized void startReadOnlyTransaction(){
      if(creationTime.get()== null){
          creationTime.set(new Integer(GlobalCounter.getNext()));
      }
}
```

Figure 5.23: Implementation of Multi-Version LockBased – Part II

The transaction commit phase of a *Multi-Version* object is handled differently (Figure 5.23). *Multi-Version* relies on *Tracked* in identifying the objects modified by the committing transaction. The sets of objects written-to and updated are obtained on lines 47 and 58 respectively. The state of each modified object is duplicated (lines 50-51, 62-63), annotated with a commit timestamp (lines 52, 64) and added to the HCS (lines 53,65) before releasing the previously acquired locks (lines 55-56, 68-69).

Read-only transactions are handled differently. They do not have to acquire transactional locks before proceeding because they are executed on the objects in the HCS. As discussed above, each object in the HCS has a timestamp during which its state was valid. Consequently, *read-only* transactions must also be assigned creation timestamps to determine the objects in the HCS on which they should be executed (lines 72 to 76 - figure 5.22). Upon the interception of a *read-only* transaction (lines 26 to 38 – figure 5.21), *Multi-Version* looks through the HCS to determine the object on which to execute the operation. The creation timestamp of the transaction is obtained on line 18, lines 28 to 35 searches for a version of the target object (i.e., *versionID*) in the history of committed states with the highest timestamp that is lower than the timestamp of the *read-only* transaction and line 37 assigns this *versionID* to the view of the invoking transaction using *Versioned.setCurrentVersion(ObjectName, versionID)*.

Discussion

Populating the HCS only at transaction commit time insinuates that *read-only* transactions on a transactional object must be preceded by at least one write or update transaction. This is not necessarily the case in practice; several *read-only* transactions may be executed on an object before a *Write* or an *Update*. To accommodate this, an *after* advice (lines 12 to 16 – figure 5.21) intercepts the creation of every *Multi-version* object, duplicates its state (line 13), annotates the duplicated state with a commit timestamp (line 14) and adds it to the HCS (line 15).

This implementation has three drawbacks. First, the objects in the HCS are accessed exclusively by *read-only* operations. *Read-only* operations do not conflict; hence the functionality of *Shared* is not necessary for the objects in the HCS. The *Shared* aspect should therefore be disabled on these objects so as to optimize system performance. As explained before, AspectJ does not support runtime disabling and re-enabling of pointcuts and the *if(Boolean)* pointcut at best, disables only an advice.

The second limitation of this implementation is the well-discussed *Weak aspect-to-class binding*. Method calls or executions of indirectly introduced methods on a *Multi-Version* object cannot be captured. Lastly, this implementation cannot capture reflective calls or executions of non-overridden inherited methods (*Reflection/Super-class method execution dilemma*). The ramifications of these deficiencies are the same as in the *LockBased* aspect.

5.3.12 Optimistic

Summary of functionality

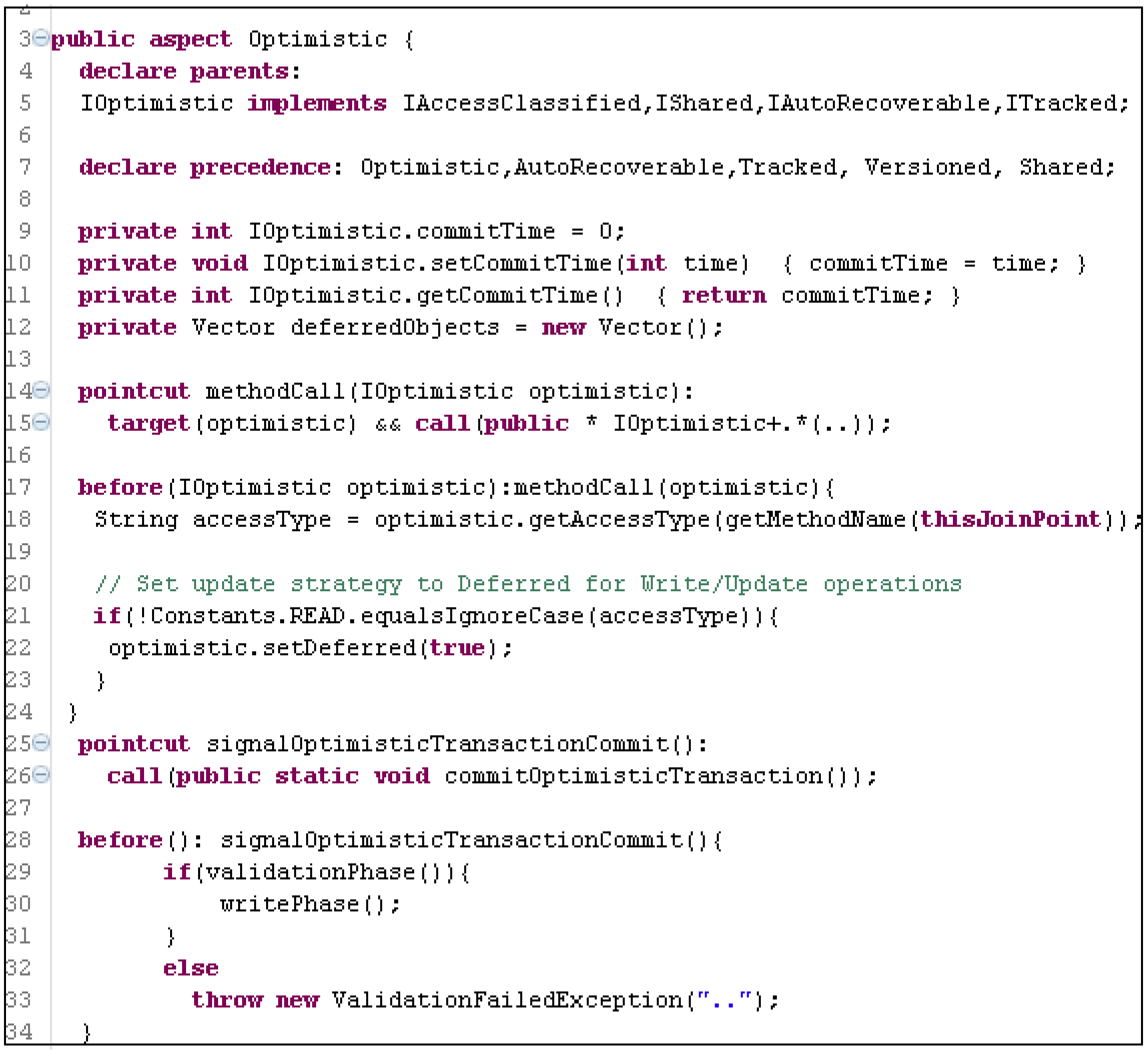
- This aspect provides support for *optimistic* concurrency control with *deferred* update and *backward* validation.
- Each concurrent *modifier* transaction works on a local copy of the state of a transactional object.
- The modifications on the local copies are made global upon successful transaction validation.

Implementation

The first part of the implementation of the *Optimistic* aspect is presented in figure 5.24. This aspect relies on *AccessClassified* (in distinguishing between *Read*, *Write* and *Update* operations), *Shared* (in preventing threads within a transaction from currently modify an object's state), *AutoRecoverable* (in creating local copies of the state of a transactional object), and *Tracked* (in identifying the transactional objects modified by a given transaction) - lines 4 to 5. The desired execution order of these aspects is specified on line 7 (i.e., *Optimistic*, *AutoRecoverable*, *Tracked*, *Versioned*, *Shared*,). The update strategy has to be set to *deferred* by *Optimistic* before *AutoRecoverable* executes; otherwise, a local version wouldn't be created in time for the transaction and the operation will mistakenly be executed on the *main version*. The object is then *Tracked*, the operation directed to the appropriate local object by *Versioned*, and mutual exclusion to the state of the object ensured by *Shared*.

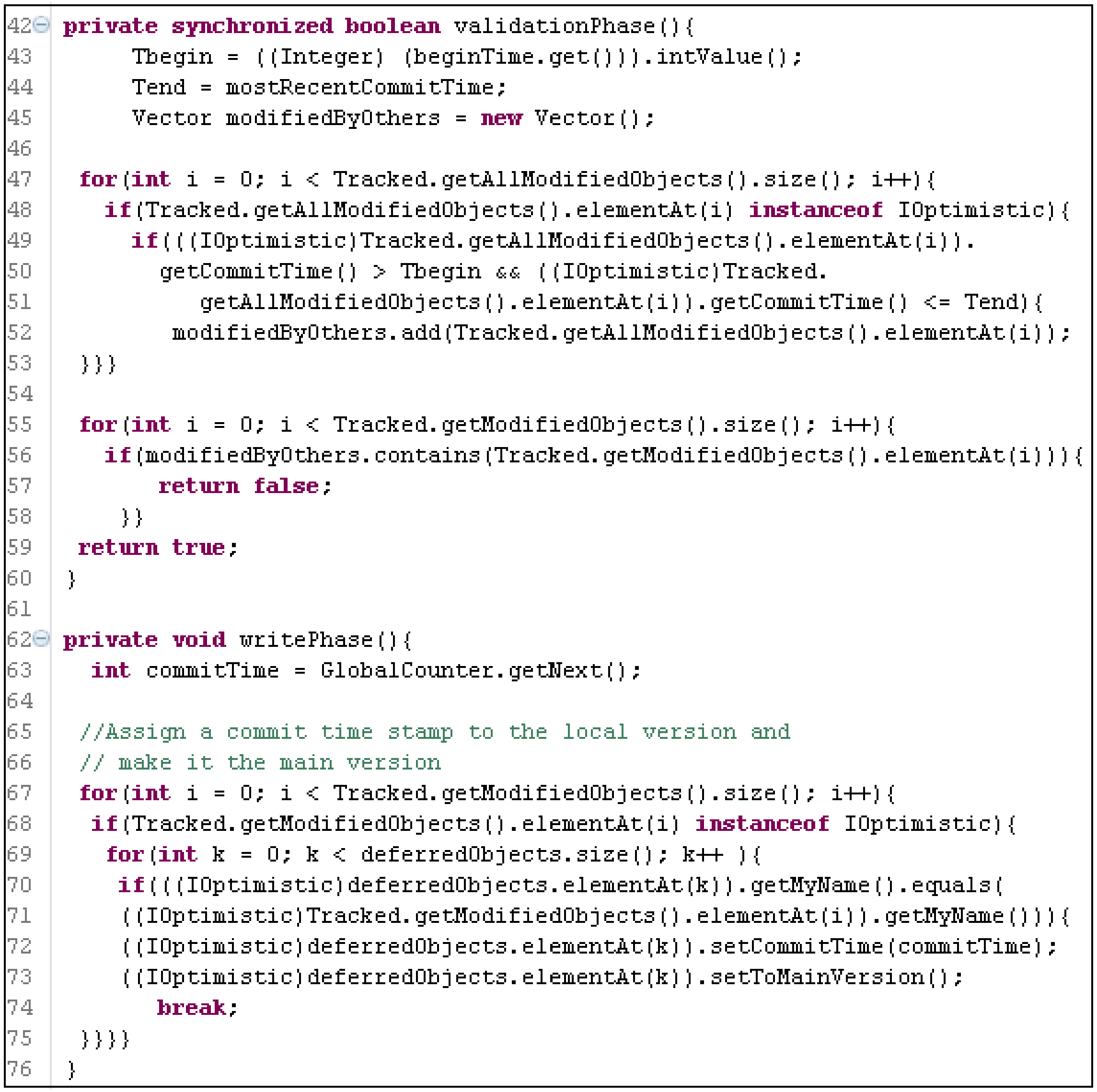
The static crosscutting behaviours (fields and methods) necessary to the support the *Optimistic* aspect are presented on lines 9 to 11. The poincut for capturing calls to an *Optimistic* object is defined on lines 14 to 15. The *before* advice (lines 17 to 24) sets the update strategy to *deferred* for *Write* and *Update* operations. This signals *AutoRecoverable* to create a local copy of the transactional object state for the current executing transaction. This process is repeated for each concurrently executing transaction, giving rise to multiple uncommitted versions of the transactional object. The update strategy for read-only transactions is *in-place* - by default. This means that read-only transactions are always executed on the *main version* (that holds the committed states) of a transactional object.

Modifier operations executed on *Optimistic* transactional objects must be validated before the changes are committed (i.e., becomes global). Transactions signal their need for validation by invoking the global method: *commitOptimisticTransaction()*. The pointcut on lines 25 to 26 intercepts the request for transaction commit, and the *before* advice (lines 28 to 34) validates the committing transaction (line 29) and commits the changes upon a successful validation (line 30).



```
 3 public aspect Optimistic {
 4   declare parents:
 5   IOptimistic implements IAccessClassified,IShared,IAutoRecoverable,ITracked;
 6
 7   declare precedence: Optimistic,AutoRecoverable,Tracked, Versioned, Shared;
 8
 9   private int IOptimistic.commitTime = 0;
10   private void IOptimistic.setCommitTime(int time)  { commitTime = time; }
11   private int IOptimistic.getCommitTime()  { return commitTime; }
12   private Vector deferredObjects = new Vector();
13
14   pointcut methodCall(IOptimistic optimistic):
15     target(optimistic) && call(public * IOptimistic+.*(..));
16
17   before(IOptimistic optimistic):methodCall(optimistic){
18    String accessType = optimistic.getAccessType(getMethodName(thisJoinPoint));
19
20    // Set update strategy to Deferred for Write/Update operations
21    if(!Constants.READ.equalsIgnoreCase(accessType)){
22     optimistic.setDeferred(true);
23    }
24  }
25   pointcut signalOptimisticTransactionCommit():
26     call(public static void commitOptimisticTransaction());
27
28   before(): signalOptimisticTransactionCommit(){
29         if(validationPhase()){
30             writePhase();
31         }
32         else
33           throw new ValidationFailedException("..");
34   }
```

Figure 5.24: Implementation of Optimistic – Part I



```
42  private synchronized boolean validationPhase(){
43       Tbegin = ((Integer) (beginTime.get())).intValue();
44       Tend = mostRecentCommitTime;
45       Vector modifiedByOthers = new Vector();
46
47   for(int i = 0; i < Tracked.getAllModifiedObjects().size(); i++){
48     if(Tracked.getAllModifiedObjects().elementAt(i) instanceof IOptimistic){
49       if(((IOptimistic)Tracked.getAllModifiedObjects().elementAt(i)).
50         getCommitTime() > Tbegin && ((IOptimistic)Tracked.
51            getAllModifiedObjects().elementAt(i)).getCommitTime() <= Tend){
52          modifiedByOthers.add(Tracked.getAllModifiedObjects().elementAt(i));
53   }}}
54
55   for(int i = 0; i < Tracked.getModifiedObjects().size(); i++){
56     if(modifiedByOthers.contains(Tracked.getModifiedObjects().elementAt(i))){
57         return false;
58      }}
59   return true;
60  }
61
62  private void writePhase(){
63    int commitTime = GlobalCounter.getNext();
64
65   //Assign a commit time stamp to the local version and
66   // make it the main version
67   for(int i = 0; i < Tracked.getModifiedObjects().size(); i++){
68    if(Tracked.getModifiedObjects().elementAt(i) instanceof IOptimistic){
69     for(int k = 0; k < deferredObjects.size(); k++ ){
70      if(((IOptimistic)deferredObjects.elementAt(k)).getMyName().equals(
71       ((IOptimistic)Tracked.getModifiedObjects().elementAt(i)).getMyName())){
72       ((IOptimistic)deferredObjects.elementAt(k)).setCommitTime(commitTime);
73       ((IOptimistic)deferredObjects.elementAt(k)).setToMainVersion();
74         break;
75   }}}}
76  }
```

Figure 5.25: Implementation of Optimistic – Part II

Figure 5.25 presents a stripped-down version of the implementation of the *validation* and *writes* phases of a transaction. Using *Tracked*, the *validation phase* (lines 42 to 60) first obtains the set of objects modified by all transactions (excluding the validating transaction) between *Tbegin* (i.e., the start time of the transaction) and *Tend* (i.e., timestamp of the most recently committed transaction) – lines 47 to 53. Validation is successful if the intersection set between the aforementioned set of modified objects and the set of objects modified by the validating transaction (*Track.getModifiedObject()*) is empty; otherwise, validation is unsuccessful – lines 55 to 59. The *write phase* (lines 62 to 75) assigns a commit timestamp to each of the local versions of the committing transaction before making these versions the *main versions* of the respective objects.

Discussion

The *Optimistic* aspect suffers from *Weak aspect-to-class binding* and *Reflection/Super-class method execution dilemma*.

Chapter 6 ~ Encountered AspectJ Limitations and Possible Improvements

In chapter 5, I discussed the implementation of the aspects proposed in the case study and highlighted the encountered AspectJ limitations. This chapter provides an in-depth discussion of these limitations, possible work-around solutions (where possible), and potential improvements to the AspectJ language features (where appropriate).

6.1 Lack of support for runtime disabling and re-enabling of pointcuts

Limitation

Aspects are statically deployed in AspectJ; i.e., the crosscutting behaviours specified in the aspects become effective in the base applications once they are woven together and these crosscutting behaviours cannot be altered at runtime. For instance, assuming that the *Account* class is *IShared* and we have an *Account* object (*AccountObject*) that is accessed only by *observer* operations (such as in the history of committed states – section 3.2.2), the *Shared* aspect would unnecessarily obligate these operations to obtain a *read* lock on *AccountObject* before proceeding even though these operations do not conflict. Ideally, we should be able to disable the *Shared* aspect of *Account* objects that are accessed only by *observer* operations so as to maximize system performance. This limitation is not unique to transaction frameworks. Given that an aspect attached to an object consumes significant system resources, an application server may want to disable this aspect when the object is cached and re-enable it upon a request for the object.

The *if(BooleanExpression)* pointcut of AspectJ is often promoted as the construct for achieving runtime disabling and re-enabling of aspects but this claim is only partially true. This pointcut is typically used to determine whether the advice(s) to be applied at a *target-joinpoint* should be executed. This implies that the *target-joinpoint* will always be intercepted but the execution of its associated advice is conditional on the value of *BooleanExpression*. Consequently, the *if(BooleanExpression)* pointcut can only be used in disabling or enabling advices not aspects as it is believed. The use of this pointcut in our case (i.e., on objects in the history of committed states) implies that *read* locks are no longer acquired before executing *observer* operations; however, every *observer* operation invoked on *AccountObject* would still unnecessarily be intercepted by *Shared*. The performance overhead incurred by the execution of an *observer* operation is therefore unnecessary increased by the runtime static check. These performance overheads can easily add up in read-dominant applications.

Possible solution(s)

Ideally, the performance overhead for executing an *observer* operation on a *Shared Account* object in the history of committed states should be comparable to the performance overhead of executing the same *observer* operation on an *Account* object that does not implement *IShared*. To achieve this, AspectJ would have to provide new program constructs to support runtime disabling and re-enabling of the pointcuts within an aspect on a per-object basis. This would eliminate the need for the *if(BooleanExpression)* pointcut and the performance overhead associated with it since an advice cannot be executed once its associated pointcut(s) are disabled.

I propose the addition of two static aspect methods: *enablePointcut(PointcutPattern)* and *disablePointcut(PointcutPattern)*. The assumption is that all pointcuts (both *named* and *anonymous*) within an aspect are originally enabled and only *named* pointcuts can be disabled or enabled at runtime. The *disablePointcut(PointcutPattern)* static method should support runtime disabling of *named* pointcuts that matches the pattern *PointcutPattern*. For instance, the following statement: *Shared.aspectOf(AccountObject).disablePointcut(PointcutPattern)* should disable all the pointcuts with name matching the pattern *PointcutPattern* within the instance of the *Shared* aspect associated with the *AccountObject*. That is, the join points that were to be captured by the pointcut(s) of pattern *PointcutPattern* in the instance of the *Shared* aspect associated with the *AccountObject* should no longer be intercepted. The static method *enablePointcut(PointcutPattern)* should support runtime re-enabling of previously disabled pointcuts in a similar manner.

6.2 Weak aspect-to-class binding

Limitation

Abstract introduction (also known as *indirect introduction*) [19, 20] has been proposed as a strategy for implementing reusable static crosscutting behaviours that can be used in different contexts. It allows us to "*collect several extrinsic properties from different perspectives within one unit and defers the binding to existing objects*" [19]. As explained in section 5.2, the target unit (also known as *introduction container*) can either be a *class* or an *interface* in AspectJ. A *Class* cannot be effectively used as an *introduction container*; firstly, because multiple-inheritance is not supported in Java and secondly, because a *class* cannot be an ancestor to an *interface*. In our case, these extrinsic static crosscutting behaviors were collected within dummy interfaces (via the *inter-type member introduction* concept of AspectJ) and these interfaces were later bind to application classes using the *declare parents* construct. For instance, declaring the *Account* class as implementing *ICopyable* introduces two additional public operations: *replaceState(SourceObject)* and *clone()* into every *Account* object to support state replacement and cloning respectively.

Assuming that the *Account* class also implements *IShared* (such as in *LockBased*), it is logical to assume that the call and execution of *Account.replaceState(SourceObject)* would be captured by the pointcuts *call(public * IShared+.*(..))* and *execution(public * IShared+.*(..))* respectively of the *Shared* aspect since the method *replaceState(SourceObject)* was actually introduced into the *Account* class and the *Account* class is *IShared*. This is unfortunately not the case because the actual call and execution join points of the *replaceState(SourceObject)* method are *call(ICopyable.replaceState(..))* and *execution(ICopyable.replaceState(..))* respectively. That is, AspectJ associates the call and the execution join points of indirectly introduced methods with the *introduction container* instead of the application class (*Weak aspect-to-class binding*). In the terminology of [20], AspectJ binds the self-reference *this* of the indirectly introduced methods to the container type (*ICopyable*) and not the target class (*Account*). Therefore, there is the possibility of cloning or replacing the state of the *Shared Account* object through the indirectly introduced operations (i.e., *clone()* and *replaceState(..)*) while it is being modified by a different concurrent transaction. The use of *indirect introduction* as a means for providing default interface implementation although effective is therefore not equivalent to its

Java counterpart (i.e., implementing the methods inherited from super interfaces within the subclasses).

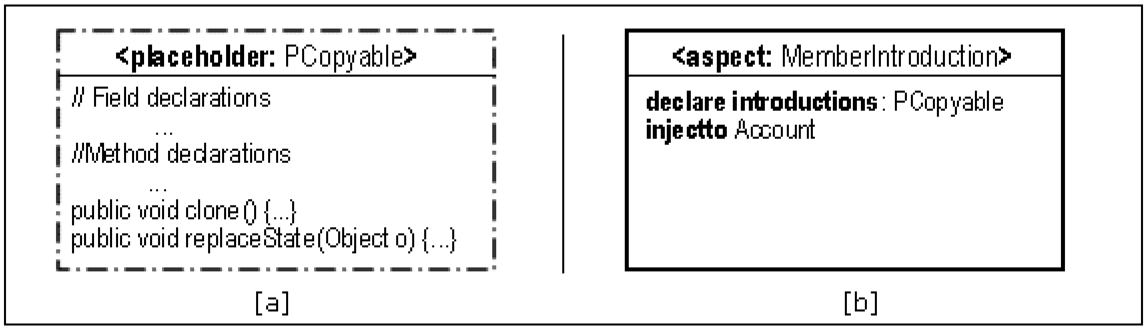
The same can be said for other aspects such as *AutoRecoverable* and *Tracked*. Transactional objects by nature assume multiple roles as they move through the execution of a transaction. For instance, a *LockBased AccountObject* would assume the roles of *IShared*, *ICopyable*, *IAutoRecoverable* and *ITracked* from the perspective of the corresponding aspects. It is therefore desirable that the call and execution join points of the operations (be it *declared*, *inherited* or *introduced*) of a transactional object be associated with the object instead of the *introduction container* so as to effectively enforce the ACID properties of transactions. It is worth noting that this deficiency is not restricted to transaction frameworks only. Any aspect-oriented framework that works at the granularity of methods is a potential victim to the *weak aspect-to-class* binding problem.

Possible solution(s)

A crude work-around solution for this problem is to declare the *ICopyable* interface as implementing *IShared* using the *declare parents* construct of AspectJ. This would obligate the *replaceState(SourceObject)* and *clone()* methods of the *AccountObject* to acquire the appropriate lock from the *Shared* aspect before executing. A similar work-around solution can be created for the other aspects (such as *Tracked*, *AutoRecoverable* and *Versioned*) that share this deficiency. Although this work-around solution addresses this deficiency, it is too specific to be generalized (i.e., it is not effectively reusable) and often results in a complex inheritance hierarchy.

The *Weak aspect-to-class binding* problem can be traced back to the weaving process of *indirect introductions* in AspectJ. These bindings are performed at method introduction time (i.e., as soon as the methods are introduced into the dummy interfaces) when the target classes are not yet known; hence, the association of the self-reference *this* with the interfaces instead of the target classes. Although AspectJ effectively collects these extrinsic static crosscutting behaviours within an interface, it fails in deferring the bindings of member functions to the target classes. An ideal solution for this problem would therefore involve deferring the binding of the self-reference *this* of indirectly introduced methods to the time when all the target classes are known. This would ensure that the call and execution join points of indirectly introduced methods occur at the target classes and not the introduction container - interfaces.

The interface construct is inherited from the underlying language of AspectJ (i.e., Java); hence, modifications to the binding process may not be backward compatible. This highlights a need for an AspectJ-specific abstraction that will effectively support the reuse of static crosscutting behaviours and eliminate the *Weak aspect-to-class binding* problem. I propose the addition of a new class-like construct called a *placeholder*. As opposed to classes and interfaces, the fields and methods should not be structurally bound to the *placeholder;* that is, its functionality should exclusively be to hold static crosscutting behaviours that would later be injected into the target classes at weave-time. A *placeholder* should not be instantiable and should never have a super-class, super-interface or be part of the inheritance hierarchy.



<placeholder: PCopyable>
// Field declarations
...
//Method declarations
...
public void clone() {...}
public void replaceState(Object o) {...}
<aspect: MemberIntroduction>
declare introductions: PCopyable
injectto Account
[a]
[b]

Figure 6.1: Proposed language construct: *placeholder*

Figure 6.1[a] shows a potential declaration of a *placeholder* for the *Copyable* aspect and figure 6.1[b] suggest a new construct for injecting and binding the fields and methods to the target class. As aforementioned, *direct introduction* associates the call and execution join points of fields and methods with the target classes. The *declare introductions* construct should therefore mimic the activity of a developer in directly introducing fields and methods into the target classes. The net effect of this approach should be equivalent to a *direct introduction* of the static crosscutting behaviours into the target classes. The only difference is that the *weaver* would be responsible for performing the introduction of the methods and fields into the target classes instead of the developer – eliminating the *weak aspect-to-class binding* problem. In addition, the *placeholder* makes these behaviours reusable; hence, the use of an interface as an introduction container is no longer necessary.

The *placeholder* concept may sound much like mixins [25] but it is fundamentally different. In mixins, the call and execution of a mixin method is delegated to the mixin class not the target class. This implies that the call and the execution joint points of such methods occurs on the mixin class not the target class - *Weak aspect-to-class binding*. Consequently, mixin is actually functionally equivalent to the *indirect introduction* idiom not the *placeholder* concept. In addition, unlike mixins, the *placeholder* never becomes part of the inheritance hierarchy of the target class.

6.3 Lack of support for explicit inter-aspect configurability

Limitation

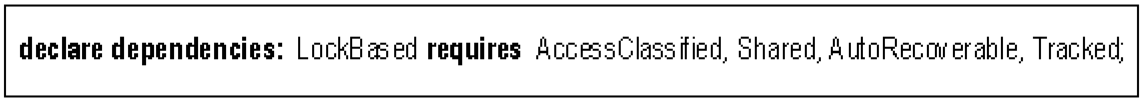
The proposed case study exhibits complex aspect dependencies and interferences. As demonstrated in chapter 3, some aspects cannot function properly without the functionality offered by other aspects. For instance, the *LockBased* aspect depends on the functionalities of the *AccessClassified* aspect (in distinguishing between *Read*, *Write* and *Update* operations), *Shared* aspect (in preventing threads within a transaction from currently modify an object's state), *AutoRecoverable* aspect (in facilitating the undo functionality - in event of a transaction abort) and the *Tracked* aspect (in keeping track of the objects accessed by a given trasaction) in order to effectively implement a pessimistic lock-based concurrency strategy. AspectJ has no construct that enables aspects to express these dependencies (i.e., inter-aspect configurability). Support for this functionality is essential to the implementation of reusable aspect-oriented frameworks. Ideally, aspects should be able to express their need of the functionality offered by other aspects while preserving obliviousness (or at least, minimize aspect-coupling).

Possible solution(s)

A work-around solution for the inter-aspect configurability problem involves associating a dummy interface with each of the aspects proposed in the case study. For instance, the interface *IShared* is associated to the aspect *Shared*, *IAutoRecoverable* is associated to

AutoRecoverable and so on. Each aspect is then implemented to apply its functionality to all the classes that implements its associated interface (e.g., the *Shared* aspect is applied to all classes that implements the *IShared* interface). Aspects express their needs for other aspects by having their associated interface implement the interfaces of the aspects they depend on. For instance, inter-aspect configurability was achieved in the *LockBased* aspect by having *ILockBased* implement *IAccessClassified*, *IShared*, *IAutoRecoverable*, *ITracked*. This enables the *LockBased* aspect to configure the *Shared*, *AccessClassified*, and *AutoRecoverable* aspects on all *ILockBased* objects.

An obvious deficiency of this work-around solution is the introduction of a complex inheritance hierarchy. In addition, the aspect dependency relationship is non-hierarchical and therefore counterintuitive to represent as an inheritance hierarchy. This highlights a need for an AspectJ specific inter-aspect configurability solution. A major challenge is how to express aspect dependencies while preserving inter-aspect obliviousness (or at least, minimize aspect-coupling).



```
declare dependencies: LockBased requires AccessClassified, Shared, AutoRecoverable, Tracked;
```

Figure 6.2: Proposed “declare dependencies” construct

Inter-aspect configurability can be expressed as proposed in figure 6.2 but achieving its ideal functionality is non-trivial. Naïvely, the advice of the *AccessClassified*, *Shared*, *AutoReciverable* and *Tracked* should be applied to all the join points pick out by *LockBased*. Ideally, *LockBased* should be able to selectively decide where each of the aspects it depends on is to be applied and the order in which they are to be executed.

6.4 Reflection/Super-class method execution dilemma

Limitation

The enforcement of the ACID properties of transactional objects occurs at the level of method invocations. To achieve these, the AspectOPTIMA framework must be able to intercept every method invocation (both *reflective* and *non-reflective*) on a transactional object and perform the appropriate pre and post actions. AspectJ provides two pointcuts for intercepting the call and execution of a method: *call(MethodPattern)* and *execution(MethodPattern)*.

The method call pointcut - *call(MethodPattern)* - would intercept *non-reflective* calls to *declared* and *inherited* methods of an object but not *reflective* calls to these methods. For instance, the pointcut *call(public * SavingAccount.*(..))* would intercept the method call *SavingAccount.debit(0)* but not *debit.invoke(SavingAccountObject , parameters)* – a reasonable conscious design decision made by the AspectJ team not to "*delve into the Java reflection library to implement call semantics*" [9]. Consequently, concurrent reflective modifications on an object cannot be prevented and the modifications made through reflective calls on the transactional objects of an aborting transaction cannot be undone - placing the system in an inconsistent state.

The method execution pointcut - *execution(MethodPattern)* – is typically used to address this deficiency. This pointcut on the other hand would intercept the execution (both *reflective* and *non-reflective*) of *declared* and *overridden-inherited* methods of an object but not the execution (both *reflective* and *non-reflective*) of *non-overridden-inherited* methods because their execution join points occur in their respective super classes. For instance, the pointcut *execution(public * SavingAccount.*(..))* would intercept both the *reflective* and *non-reflective* execution of *SavingAccount.debit(0)* but not *SavingAccount.getBalance()* because the *getBalance()* method was not overridden in *SavingAccount*; hence, its execution join point occurs in *Account* and not *SavingAccount*. As stated before, the ramifications of this deficiency may be costly.

Composing the *call(MethodPattern*) and *execution(MethodPattern*) pointcuts with an OR operator is neither a feasible solution because reflective invocations of *getBalance()* would still not be intercepted. Developers must therefore decide between exploiting the code reuse benefits of inheritance (i.e., by not unnecessarily overriding inherited implementations) or capturing reflective method executions but not both.

Possible solution(s)

As explained above, a naïve solution would be to manually override all the inherited methods from the super classes in the subclasses. This implies relinquishing the code reuse benefit of inheritance - making this solution undesirable.

A second solution would be to use a pointcut – *target(SavingAccount) && execution(public * Account+.*(..))* - that intercepts the execution of all the methods of an instance of an *Account* object when the *target* is *SavingAccount*. This pointcut will intercept both *reflective* and *non-reflective* executions of *SavingAccount.getBalance()* and *SavingAccount.debit(0)* because *SavingAccount* is a subclass of *Account*. In addition, we do not have to worry about unnecessarily intercepting the executions of operations on *CheckingAccount* objects because the *target* of the pointcut is *SavingAccount*. Although this solution solves the problem, it is application-specific and cannot be reused in a generic context. In order words, we have to know precisely what the name of the *target* subclass is to avoid the erroneous application of advices to other subclasses of the same super class. This solution is therefore not applicable in our case because our objective is to implement a reusable aspect-oriented ACID framework for transactional objects.

As demonstrated, none of these work-around solutions adequately satisfy our needs. This expresses a need to reconsider the design decision made by the AspectJ team not to intercept reflective method calls. Support for this functionality is essential in implementing a reusable and robust aspect-oriented framework. I propose the addition of an *inheritance-conscious* method execution pointcut: *superexecution(MethodPattern)*. Given a class with no super-classes (e.g., *Account*), this pointcut should behave exactly as the *execution(MethodPattern)* pointcut (i.e., it should intercept both *reflective* and *non-reflective* execution of *declared* methods). When used on a class with super-classes (e.g., *SavingAccount*), it should signal a need for the *weaver* to inline stub-methods with calls to their corresponding *non-overridden* inherited methods (i.e., *public double getBalance()* – in this case) within the body of the target-class (i.e., *SavingAccount*). This ensures that the execution join points of *non-overridden* inherited methods occur in the target sub-classes - eliminating the reflection/super-class method execution dilemma problem. The space requirement of this approach is not significant since each non-overridden method will need just a stub with a single delegation call.

6.5 Lack of support for per-object association of aspects

Limitation

An often-desired functionality in complex aspect-oriented frameworks (e.g., distributed and multi-user systems) is the ability to selectively apply different aspects to different objects of the same class. For instance, one might want to have half of the *Account* objects be *LockBased,* and the other half *Optimistic*. This requirement is fundamentally different from the need of runtime disabling and re-enabling of pointcuts (section 6.1) because in this case, we either apply an entire aspect to an object of a class or not, and the pointcuts of an aspect that wasn't applied to an object cannot be later enabled at runtime.

As explained before, aspects are statically deployed in AspectJ; therefore, the crosscutting behaviours become effective in every object of a class once the base application is woven together with the aspects. Therefore, the current AspectJ implementation does not permit a developer to selectively decide at runtime the objects of a class to which an aspect should be applied.

Possible solution(s)

Support for per-object association of aspects should not eliminate per-class association of aspects currently supported by AspectJ. Developers should be given the option to decide on a per-aspect basis the desired association. A potential solution for per-object association of aspects might involve splitting the aspect application process in AspectJ into two phases: the *preparation phase* and the *activation phase*.

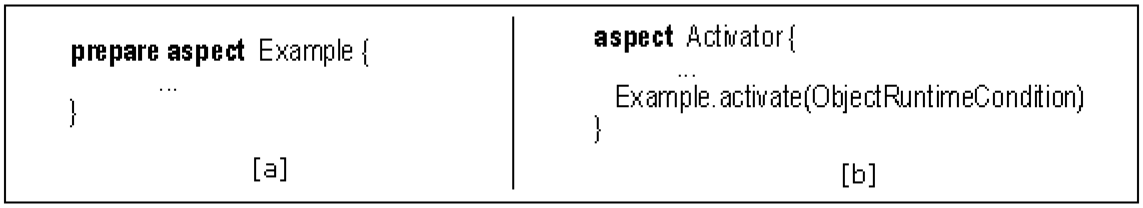
Preparation Phase

The *preparation phase* should occur at compile-time. Potential per-object aspects should be tagged with the keyword “*prepare*” as shown in figure 6.3[a]. During this phase, the crosscutting behaviours specified in the aspects should be applied in a *dormant* fashion to the respective join points of interest in the base classes. Being *dormant* implies that the join points of interest are byte-code instrumented but the functionalities of the aspects are not currently available to the objects of the base application. That is, without explicit activation, the execution of the application would be unaffected by the *dormant* aspects.

Activation Phase

The *activation phase* of the *dormant* aspects should occur at runtime and on a per-object basis. AspectJ should provide a program construct to support the activation of any (or all) of the *dormant* aspects (applied to a given class during the *preparation phase*) on a per-object basis at runtime. This phase therefore enables developers to selectively decide whether the functionalities of an aspect should be made available to a given object of a previously prepared class. Obviously, only aspects that were applied to a given class during the *preparation phase* can be activated at runtime.

An obvious concern of this approach is a compromise of the aforementioned oblivious property of aspect-oriented programming. This is because the base application may have to be aware of the aspects advising it in order to decide which aspects of a prepared object should be activated. The primary objective of aspect-oriented programming is to modularize crosscutting concerns; hence, partial compromise of obliviousness can be overlooked. If need be, previously prepared aspects could be activated by other aspects – figure 6.3[b]. These aspects will obviously require some runtime information (*ObjectRuntimeCondition*) about the objects to determine whether or not they should be activated.



```
prepare aspect Example {
        ...
}
```

[a]

```
aspect Activator {
        ...
   Example.activate(ObjectRuntimeCondition)
}
```

[b]

Figure 6.3 : Proposed “prepare” construct and activation

Chapter 7 ~ Related Work

The identification of AspectJ limitations or its application in the context of concurrency control and recovery is not unique to this thesis. I present some of the works related to this thesis in this section and also discuss how other AOP tools have attempted to address some of the limitations of AspectJ.

7.1 Other AOP tools

CaesarJ

In [7, 23], Mezini et al. identified several deficiencies of AspectJ's join point interception model, namely:

- *Lack of support for sophisticated mappings*: the authors argued that the mapping from aspect abstractions to base classes via the *declare parents* construct is effective only when each aspect abstraction has a corresponding base class. Using examples, they demonstrated the deficiency of AspectJ in handling sophisticated mappings that deviate from the norm.
- *Lack of support for reusable aspect bindings*: it was further argued that the aspect-to-class binding achieved via the *declare parents* construct strongly binds an aspect to a particular base class; hence, such bindings cannot be effectively reused.

- *Lack of support for aspectual polymorphism*: this limitation is comparable to the *lack of support for per-object association of aspects* identified in this thesis. The paper argued that it is currently not possible in AspectJ to determine at runtime whether an aspect should be applied or not, or which implementation of the aspect to apply.

The authors then proposed a new aspect-oriented programming tool called *CaesarJ* [23] to address these deficiencies. *CaesarJ* is based on Aspect Collaboration Interfaces (ACI). In ACI, the aspect implementation is decoupled from the aspect binding, with each defined in an independent but indirectly connected module. This tool relies on a new type called *weavelet* for composing the implementation and the binding of the aspect into a final system. Different *weavelets* can combine an aspect binding with different aspect implementations, and different *weavelets* can also be used in combining a particular aspect implementation with different aspect bindings; making both the aspect bindings and implementations independently reusable. As opposed to *AspectJ*, compiling these bindings (i.e., *weavelet*) with the base application does not have any effect on the execution of the application. This is because the *weavelets* must be explicitly deployed to activate their pointcuts and advices. These *weavelets* can also be deployed statically or dynamically; hence, the support for runtime deployment of aspects on a per-object basis. Although *CaesarJ* looks like our ideal AOP tool, it shares some of the deficiencies of AspectJ identified in this thesis. For instance, the pointcut(s) of a deployed *weavelet* cannot be disabled and later re-enabled at runtime on a per-object basis.

JBossAOP

JBossAOP [5, 26] is another aspect-oriented programming tool in my opinion which comes close to addressing the encountered limitations discussed in this thesis. It supports both per-instance association of aspects and hot deployment of aspects (i.e., the ability to unregister existing advice bindings – pointcuts - and deploy new bindings to previously instrumented join points at runtime). This dynamism is accomplished using the "*prepare*" statement of JBossAOP which instruments target join points so that pointcuts and advices can be later applied at runtime. Notwithstanding, this tool suffers from the *weak aspect-to-class* binding problem. This is because the implementation of reusable static crosscutting behaviours can only be achieved through mixins in JBossAOP. This implies that the call and execution of a mixin method is delegated to the mixin class, hence, the call and execution join points of such methods are associated with the mixin class not the target class.

7.2 Other Concurrency Control and Persistence frameworks

Cunha et al. [24] explored the possibility of implementing a reusable aspect-oriented implementation of concurrency control patterns and mechanisms for Threads in AspectJ. The authors illustrated how abstract pointcut interfaces and annotations (newly introduced in Java 1.5) can be used in implementing one-way calls, synchronization barriers, reader/writer locks, scheduler, active objects and futures. The paper also compared the performance overhead, reusability and the (un)pluggability between conventional object-oriented implementations and AOP implementations. It was concluded that the AspectJ implementation was more reusable and pluggable but incurs a noticeable performance overhead. In addition, the authors argued that

AspectJ has a limitation in acquiring local join point information in concrete aspects because the abstract pointcuts presets the contextual information available to its sub-aspects. The work of Cunha et al. differs from that of this thesis in the amount of effort required to harness the framework's functionality. In [24], developers must provide concrete pointcuts for each of the abstract pointcuts to have their applications advised. This implies that developers are not completely oblivious of the inner workings of the framework; a luxury that is not always possible (e.g., some third-party software libraries provide only byte codes or executables). Conversely, the only work required by developers to acquire the functionality provided by the aspects in this thesis is to bind their application classes to the appropriate aspects via the *declare parents* construct. This requires no knowledge of the inner workings of the framework and can be accomplished even if the source code isn't available because *AspectJ* supports byte code weaving.

Rashid et al. [15] also explored the possibility of implementing a reusable and oblivious aspect-oriented framework for persistence in AspectJ. Using a database application as an example, the authors demonstrated incrementally how reusable aspects for database connections, data storage and updates, data retrieval and data deletion can be implemented. Unlike the *Persistence* aspect in this thesis that relies on other well-defined reusable aspects (*Serializeable*, *Copyable* and *Named*), their implementation of persistence relies on other database specific aspects that cannot be reused in a non-database persistent context. Similar to the work of Cunha et al., developers must also provide concrete pointcuts for each of the abstract pointcuts in the persistence framework to have their applications advised - hindering obliviousness.

Chapter 8 ~ Conclusions and Future Work

8.1 Conclusions

This thesis had two objectives. Firstly, to ascertain whether the decomposition of *Concurrency Control* and *Recovery* (i.e., the ACID properties of transactional objects) into reusable aspects proposed in [8] (see chapter 3) can be realistically implemented and recomposed in an aspect-oriented system to provide the desired functionality. Secondly, to evaluate the adequacy of the language features of *AspectJ* in implementing a reusable framework for the ACID properties of transactional objects.

As demonstrated in chapter 5, the reusable aspects proposed in the case study can be individually implemented and later recomposed in AspectJ to achieve various *Concurrency Control* and *Recovery* strategies. The implementation was achieved by associating a dummy interface to each of the aspects in the case study and by implementing the aspects to apply their functionalities to the classes that implement their associated interface. The binding of a reusable aspect to a base class can be accomplished either through the *declare parents* construct of AspectJ or the *implements* keyword of Java. The proposed decomposition of the ACID properties of transactional objects into reusable aspects was therefore successfully implemented in *AspectJ*, notwithstanding the encountered limitations.

That said, *AspectJ's* language features were not always explicitly helpful in implementing the reusable ACID framework. The language features were found to be inadequate in certain circumstances (section 6.4 and 6.5). This thesis identified five significant limitations of the current language features of *AspectJ*, namely: lack of support for inter-aspect configurability, lack of support for runtime disabling and re-enabling of pointcuts on a per-object basis, lack of support for per-object association of aspects, weak aspect-to-class binding and reflection/super-class method execution dilemma. Work-around solutions were utilised where possible but these solutions incur unnecessary complexity and potential performance overhead in the base applications. Finally, I proposed new language constructs and concepts towards achieving an ideal AspectJ tool (see Chapter 6 for detailed discussion of these limitations and the proposed solutions).

8.2 Future Work

Anecdotal evidence suggests that the *AspectJ* implementation of the ACID properties for transactional objects (i.e., the AspectOPTIMA framework) might incur a significant performance overhead and memory footprint relative to its object-oriented counterpart – OPTIMA [11]. A reasonable future work would involve optimizing the current *AspectJ* implementation of the AspectOPTIMA framework, obtaining the performance overhead and memory footprint for both AspectOPTIMA and OPTIMA, and determining whether there is a noticeable penalty in the migration from an object-oriented platform to an aspect-oriented platform.

Another by-product of this thesis deserving further investigation is the performance cost associated with the *automatic classification* of the operations of transactional objects (section 5.3.1). As mentioned before, the efficiency of any concurrency control or recovery framework is conditional on its ability to accurately distinguish *observer* operations from *modifier* operations. Traditionally, frameworks rely on naming conventions or user-defined metadata (such as annotations in Java) in distinguishing *observer* operations from *modifier* operations - a technique susceptible to developer classification error with potentially costly ramifications. Other frameworks avoid this risk by treating every operation as a *modifier* – incurring additional performance overhead. Automatic classification is therefore ideal since it eliminates the risks associated with erroneous classification; however, it would be counter productive if its performance overhead exceeds that of frameworks that treats every operation as a *modifier*. I intend to conduct a comparative performance study of these classification strategies in the near future.

The *Versioned* aspect, *Tracked* aspect and *AutoRecoverable* aspect share a common need for a well-defined region of interest per-transaction within which certain actions (such as object accesses) are to be monitored. This is a crosscutting concern of the aspects rather than the transactional objects. In this thesis, I have focussed on the identification and implementation of aspects that crosscut objects not aspects that crosscut other aspects. A long-term goal includes the identification and modularization of concerns that crosscut other aspects.

Bibliography

[1] G. Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 1993 (2nd ed.)

[2] W. Harrison and H. Ossher. *Subject-oriented programming: a critique of pure objects.* In Proceedings of the 8th annual conference on Object-oriented programming systems, languages, and applications. Washington, D.C., United States. Pages 411 - 428. ACM Press, 1993.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. *Aspect-Oriented Programming.* In the proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP97), Finland, June 1997.

[4] AspectJ Team. *AspectJ*, May 2006. Available at http://www.eclipse.org/aspectj/.

[5] *JbossAOP*, May 2006. Available at http://labs.jboss.com/portal/jbossaop/index.html.

[6] *Spring AOP*, May 2006. Available at http://www.springframework.org/.

[7] M. Mezini and K. Ostermann*. Conquering aspects with Caesar.* Proceedings of the 2nd international conference on Aspect-oriented software development, Boston, Massachusetts, 2003. ACM Press, Pages 90 – 99.

[8] J. Kienzle and S. Gélineau. *AO Challenge - Implementing the ACID Properties for Transactional Objects*. 5th International Conference on Aspect-Oriented Software Development (AOSD'2006), Bonn, Germany, 2006.

[9] Xerox Corporation. *Frequently Asked Questions about AspectJ*. May 2006 - Available at http://www.eclipse.org/aspectj/doc/released/faq.html.

[10] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 1993.

[11] J. Kienzle. *Open Multithreaded Transactions: A Transaction Model for Concurrent Object-Oriented Programming*. Kluwer Academic Publishers, 2003.

[12] H. Kung and J. Robinson*. On optimistic methods for concurrency control.* ACM Transactions on Database Systems, Volume 6 (Issue 2), Pages 213 - 226, June 1981.

[13] M. Herlihy. *Apologizing versus asking permission: Optimistic concurrency control for abstract data types*. ACM Trans. on Database Systems, 15(1): Pages 96 -124, March 1990.

[14] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning Publications, 2003.

[15] A. Rashid and R. Chitchyan. *Persistence as an aspect*. In Proceedings of the 2nd international conference on Aspect-oriented software development, Boston, Massachusetts, USA, March 2003, Pages 120 –129.

[16] E. Gamma and K. Beck. *Junit Framework*. Available at www.junit.org.

[17] Sun Microsystems. *Java 2 Standard Edition (J2SE 5.0)*. May 2006 - Available at: http://java.sun.com/j2se/1.5.0/.

[18] Sun Microsystems. *Java Documentation - ava.lang.annotation.Inherited*. May 2006 - Available at www.java.sun.com/j2se/1.5/docs/api/java/lang/annotation/Inherited.html.

[19] S. Hanenberg and P. Costanza. *Connecting Aspects in AspectJ: Strategies vs. Patterns*. First Workshop on Aspects, Components, and Patterns for Infrastructure Software at 1st International Conference on Aspect Oriented Software Development (AOSD), Enschede, April, 2002.

[20] S. Hanenberg and R. Unland: *Parametric Introductions*, Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, Boston, MA, ACM Press, March 2003. Pages 80-89.

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

[22] E. Hilsdale and J. Hugunin. *Advice weaving in AspectJ.* In Proceedings of the 3rd international conference on Aspect-oriented software development, pages 26--35. ACM Press, 2004.

[23] M. Mezini and K. Ostermann*. CaesarJ,* May 2006. Available at www.caesarj.org.

[24] C. Cunha, J. Sobral, and M. Monteiro. *Reusable Aspect-Oriented Implementations of Concurrency Control Patterns and Mechanisms*. 5th International Conference on Aspect-Oriented Software Development (AOSD'2006), Bonn, Germany, 2006.

[25] A. Schmidmeier, S. Hanenberg, and R. Unland: *Implementing Known Concepts in AspectJ*. 3rd Workshop on Aspect-Oriented Software Development of the SIG Object-Oriented Software Development of the German Informatics Society, Essen, Germany, March 4-5, 2003.

[26] *JbossAOP Documentation*, May 2006. Available at http://docs.jboss.org/aop/1.0/aspect-framework/.