

Aspectual Decomposition of Transactions

GÜVEN BÖLÜKBAŞI

Master of Science

Department of Computer Science

McGill University

Montréal, Québec

June 2007

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Copyright ©2007 by Güven Bölükbaşı
All rights reserved

ACKNOWLEDGEMENTS

It is a pleasure to thank the many people who made this thesis possible.

Firstly, I would like to thank my supervisor, Jörg Kienzle, the director of the Software Engineering Lab at McGill University. I could not have imagined having a better advisor for my M.Sc. Throughout my thesis, he provided encouragement, good teaching and lots of good ideas. I would have definitely been lost without him.

Very special thanks go to Meaghan Worth for giving me the extra strength, motivation and love necessary to get things done. Writing this thesis would have been extremely difficult without her and Cessna's presence.

Lastly, and most importantly, I wish to thank my family, my mother Eser Bölükbaşı, my father Nejat Bölükbaşı, and my sister Gamze Bölükbaşı. They raised me, supported me, taught me, and loved me. To them I dedicate this thesis.

ABSTRACT

The AspectOPTIMA project aims to build an aspect-oriented framework that provides run-time support for transactions. The previously established decomposition of the ACID (atomicity, consistency, isolation, durability) properties into ten well-defined reusable aspects had one limitation: it didn't take into account the concerns of transaction life-time support, resulting in the creation of a cross-cutting concern among the aspects. This thesis removes the cross-cutting concern by integrating the transactional life cycle management issues such as determining the transaction boundaries, maintaining a well-defined state and managing the involvement of the participants. The integration process results in the creation of new aspects that serve as building blocks for various transactional models. The thesis also demonstrates how these base aspects can be configured and composed in different ways to design customized transaction models with different concurrency control and recovery strategies.

ABRÉGÉ

Le but du projet AspectOPTIMA est de créer un cadre orienté-aspect (AOP) qui fournit le soutien de la gestion de transactions. La décomposition des propriétés ACID (l'atomicité, la cohérence, l'isolation et la durabilité) en dix aspects réutilisables était déjà réalisée, mais elle avait une limitation : elle ne tenait pas compte des préoccupations liées au cycle de vie des transactions. Cette limitation donnait lieu à une préoccupation non-modularisée, c'est-à-dire une préoccupation dont la fonctionnalité était parsemée à travers les aspects. Cette thèse résout cette limitation en intégrant la gestion de cycle de vie transactionnelle, tel que déterminer les frontières de transaction, maintenir son état et manipuler la participation des participants, dans la conception de AspectOPTIMA. Le processus d'intégration crée de nouveaux aspects qui servent d'éléments de construction pour créer maintes modèles transactionnels variés. La thèse démontre aussi comment ces aspects peuvent être configurés et composés de façons différentes pour concevoir des modèles de transaction personnalisés utilisant de stratégies de concurrence et de récupération différentes.

LIST OF TABLES

<u>Table</u>		<u>page</u>
2-1	Conflict Table for Strict Concurrency Control	8
6-1	The Design of Performance Monitoring Example	66
6-2	The Design of Flat Transaction Model with Optimistic Concurrency .	70
6-3	The Design of Nested Transaction Model with Pessimistic Concurrency	75
6-4	The Design of OMTT Model with Pessimistic Concurrency	80

LIST OF FIGURES

<u>Figure</u>	<u>page</u>
2-1 Flat Transaction Model	13
2-2 Nested Transaction Model	14
2-3 Multithreaded Transaction Model	15
2-4 Open Multithreaded Transaction Model	17
3-1 Control Flow for LockBased Concurrency Control	26
4-1 Visualization of a Context	31
4-2 Participants performing work on behalf of a Context	32
5-1 Aspects of Object	35
5-2 Aspects of Thread	48
5-3 Aspects of Context	51
6-1 A Sample Output of the Performance Monitoring Application	67
6-2 Interaction of Aspects for Providing Synchronous Entry	68
6-3 Sample Execution Trace for Flat Transaction Model	71
6-4 Interaction of Aspects for Deferred Update Method Invocation	73
6-5 Sample Execution Trace for Nested Transaction Model	76
6-6 Interaction of Aspects for Aborting a Nested Transaction	78
6-7 Sample Execution Trace for OMTT Model	81
6-8 Interaction of Aspects for InPlace Update Method Invocation	83
7-1 A Technique to Resolve Aspect Interferences	88

8-1	Abstract Introduction Concept	95
-----	---	----

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	i
ABSTRACT	ii
ABRÉGÉ	iii
LIST OF TABLES	iv
LIST OF FIGURES	v
1 Introduction	1
1.1 Motivation	1
1.2 Summary of Contributions	3
1.3 Thesis Road Map	5
2 Fundamental Concepts	6
2.1 Transactions	6
2.1.1 ACID Properties	7
2.1.2 Concurrency Control	8
2.1.3 Recovery Techniques	10
2.1.4 Transactional Models	11
2.2 Aspect-Oriented Programming	18
3 Existing AspectOPTIMA Framework	22
3.1 Overview	22
3.2 Decomposition of Aspects	23
3.3 Recomposition of Aspects	24
3.4 Evaluation of the Framework	26
4 Extending AspectOPTIMA	28
4.1 Integrating the Transaction Life Cycle	28

4.1.1	Why a Transaction is not an Aspect of a Data Object . . .	29
4.1.2	Why a Transaction is not an Aspect of a Thread	30
4.2	The Context Idea	31
4.3	Framework with Context	32
5	New Design of AspectOPTIMA	34
5.1	Aspects of Object	35
5.1.1	Named	36
5.1.2	AccessClassified	37
5.1.3	Copyable	37
5.1.4	Lockable	38
5.1.5	Shared	40
5.1.6	Versioned	41
5.1.7	Checkpointable	42
5.1.8	Traceable	43
5.1.9	Serializable	44
5.1.10	Persistable	45
5.1.11	ContextAware	46
5.1.12	ContextTracing	47
5.2	Aspects of Thread	48
5.2.1	ContextParticipant	48
5.2.2	Collaborating	49
5.2.3	OutcomeAffecting	50
5.3	Aspects of Context	50
5.3.1	Tracing	51
5.3.2	Checkpointing	52
5.3.3	Deferring	53
5.3.4	Recovering	54
5.3.5	Two-Phase Locking	56
5.3.6	Nested	57
5.3.7	OutcomeAware	58
5.3.8	Collaborative	58
5.3.9	EntrySynchronizing	60
5.3.10	ExitSynchronizing	60
5.3.11	Pausable	61
5.3.12	Terminatable	62
5.3.13	OutcomeVoted	63

6	Using AspectOPTIMA: Reconstructing Transaction Models from Aspects	65
6.1	Performance Monitoring Context	65
6.2	Flat Transaction Model With Optimistic Concurrency Control & Deferred Update	69
6.3	Nested Transaction Model with Pessimistic Concurrency Control & InPlace Update	74
6.4	Open Multithreaded Transaction Model with Pessimistic Concurrency Control & InPlace Update	79
7	Aspect Interferences in AspectOPTIMA	86
7.1	Definition of Aspect Interference	86
7.2	Resolving Interferences in a Reusable Way	87
7.3	Interfering Aspects in AspectOPTIMA	88
8	AspectOPTIMA Implementation Comments	92
8.1	Implementation Platform	92
8.2	Achieving Reusability	94
8.3	The Context Object & Context Manager	96
8.4	Framework API	97
9	Related Work	98
9.1	ACTA Framework	98
9.2	Modularization of Advanced Transaction Management	99
9.3	Framework of Concurrency Patterns and Mechanisms	100
9.4	Persistence Framework	102
9.5	Framed Aspects	102
10	Conclusion	104
10.1	Summary of Results	104
10.2	Future Work	106
	REFERENCES	108

CHAPTER 1

Introduction

1.1 Motivation

Object-oriented programming (OOP) [Boo94] made a major impact on the process of software development when it has introduced the concepts of object abstraction, encapsulation, inheritance and polymorphism. It allows the decomposition of real world problems into *objects*, which present an appropriate layer of abstraction by encapsulating the data and the related behavior. Although object-orientation has met great success in modeling and implementing complex software systems, its weakness is in applying common behavior that spans multiple non-related object models. Attempts to implement such crosscutting concerns in object-oriented programming often results in systems that are difficult to reuse or maintain; this is where aspect-oriented programming (AOP) [KLM⁺97] comes in. AOP has been proposed as a new programming paradigm that enables the modularization of crosscutting concerns, resulting in systems that are easier to understand, maintain and reuse. AOP and OOP are not competing technologies, but actually complement each other quite nicely.

Since its initial conception in 1997, aspect-oriented programming has come a long way. However, as a common problem with all new technologies, aspect-orientation suffers from the lack of experience that would evaluate its evolution process. Hopefully, this gap is being filled by the yearly Aspect-Oriented Software Development conference and workshops such as SPLAT (Software Engineering Properties

of Languages and Aspect Technologies), FOAL (Foundations of Aspect-Oriented Languages) or DAW (Dynamic Aspects Workshop). These events now attract the attention of many researchers both from academia and the industry. Furthermore, this user community applies aspect-orientation to their respective domains and proposes new aspect-oriented language features to fit their needs. Yet, these proposed features may sometimes not be applicable to more general domains, because the examples used for justifying such features often remain to be too specialized.

The case study proposed in [KG06], promises to be a perfect candidate that may serve as a benchmark for evaluating the new AOP approaches. It offers a language-independent decomposition of the ACID properties (Atomicity, Consistency, Isolation and Durability) of transactions into reusable aspects. It also 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 concepts.

The case study was a by-product of the attempt to migrate OPTIMA [Kie03a] – a framework that provides transaction support for concurrent object-oriented programming languages – from an object-oriented to an aspect-oriented platform. The first aspect-oriented design and implementation of the framework successfully modularized concurrency control and recovery concerns. However, other concerns, such as transaction life-cycle management, were only rudimentarily supported and poorly modularized, and as a result their functionality crosscut the design of the other aspects in the framework. Motivated by this observations, this thesis focusses on a re-design and extension of the AspectOPTIMA framework to address this limitation.

A summary of the main contributions and the limitations they address is presented in the following subsection:

1.2 Summary of Contributions

The main contributions of this thesis are the following:

- The transactional life-cycle management concerns have been integrated into the AspectOPTIMA framework.

The first version of the framework [KG06] did not take into account the concerns that are related to transaction life-time support. As the authors pointed out, such a limitation results in a crosscutting concern among the aspects of the framework. The thesis removes this crosscutting concern by integrating the transactional life-cycle management issues into the framework. These issues consist of determining the transaction zones/boundaries, keeping the state of a transaction, monitoring nesting relationships among transactions and controlling the involvement of the participants.

As a part of the integration process, the thesis shows why neither the object nor the thread entity would be appropriate entities to encapsulate the concerns listed above. Therefore, a new central entity is proposed, called a context, which represents a region of computation with a well-defined state and behavior. The introduction of the context concept required a major refactoring of the framework. New aspects, which bind the object and thread to their context entity have been added, and some of the existing aspects have been modified.

- The AspectOPTIMA framework has been re-designed to include new aspects that enable the construction of various transactional models.

The introduction of the core transactional life-cycle concerns opens the door for new advances in the AspectOPTIMA framework. Initially, the thesis examines a variety of transactional models, trying to identify the common points in them. Later, these shared functionalities are encapsulated in reusable aspects that are now included in the AspectOPTIMA framework. The thesis provides examples of how various configurations and compositions of these aspects allow for the creation of customized transactional models.

- A new technique has been proposed for resolving the aspect interferences in a reusable manner.

The new design of the AspectOPTIMA framework includes 28 different reusable aspects. Naturally, some of these aspects conflict with each other, and the composition of such aspects results in interferences. By identifying and analyzing these conflicting aspects, the thesis comes up with a new method for solving the aspect interferences. The interferences in the AspectOPTIMA framework implementation are resolved by introducing new aspects that address the conflicts of the other aspects, if they are applied to the same object / thread / context.

- As a technical contribution, the new design of the AspectOPTIMA framework has been implemented.

In order to verify the correctness of the design, the thesis also includes an implementation of the new AspectOPTIMA framework. The implementation is based on AspectJ [KHH⁺01], probably the most mature and well-known aspect-oriented language.

1.3 Thesis Road Map

The rest of the thesis is organized as follows:

Chapter 2 presents the background information relevant to the thesis. It covers the fundamentals of transactions and aspect-oriented programming as it applies in the context of this thesis.

Chapter 3 summarizes the existing AspectOPTIMA framework, which is a case study [KG06] carried out by Software Engineering Lab at McGill University. It gives an overview on the design of the framework, and also provides a brief evaluation of it.

Chapter 4 gives a high level explanation of how the thesis plans to extend the AspectOPTIMA framework.

Chapter 5 presents the new design of the AspectOPTIMA framework. The motivation, functionality and reusability of each aspect is described in detail.

Chapter 6 explains how the aspects of the framework can be re-composed to build various models. It illustrates the composition and execution behavior of four different models that were designed using the aspects of the framework.

Chapter 7 introduces the aspect interference concept. It proposes a novel technique to resolve these interferences and applies this technique on the interfering aspects of the framework.

Chapter 8 provides some details on the implementation of the AspectOPTIMA framework.

Chapter 9 presents the related work.

Chapter 10 contains the conclusions of the thesis, as well as a future work section.

CHAPTER 2

Fundamental Concepts

The thesis incorporates ideas from two different domains: Transactions and Aspect-Oriented Programming. This chapter presents background information on these two research topics. Section 2.1 introduces the fundamentals of transactions (ACID properties, concurrency control, recovery techniques) and gives a brief overview of extended transactional models. The main concepts of aspect-oriented programming are introduced in section 2.2.

2.1 Transactions

A transaction [GR93] is a construct that conceptually groups a set of individual operations together to form a single, consistent high-level operation. The individual operations in a transaction either succeed or fail as a group, and appear indivisible with respect to concurrently executing transactions. A simple example of a transaction is a *bank transfer*, which usually is composed of two individual operations: A *withdraw* operation from one account and a *deposit* operation to another account. Conceptually, to complete a successful transfer, both operations must be executed. Partial execution, for instance only a withdraw, would lead to an inconsistent state.

There are three standard procedures that mark transaction boundaries: *begin*, *commit* and *abort*. *Begin* marks the initiation of a transaction, implying that all subsequent operations are to be performed on behalf of this transaction. At any point during the execution of the transaction it can be aborted, which means that all the

changes to data objects made by that transaction must be undone. A commit signals the successful completion of a transaction, allowing the effects of the transaction to become permanent and visible to the outside.

2.1.1 ACID Properties

Transactions enforce several properties that are defined as: atomicity, consistency, isolation and durability (ACID) [GR93].

- **Atomicity:** The atomicity property refers to the ability to guarantee that either *all* operations of a transaction are performed or *none* of them are. In other words, atomicity makes sure that the execution of only a subset of operations is not allowed.
- **Consistency:** The consistency property states that a transaction must preserve the integrity of the data within the system. Ensuring this property is largely the responsibility of the application developer, who has to make sure that the set of operations as a whole within each transaction move the system to a new consistent state. That way, if the initial state of the system is consistent, then all future system states will be as well.
- **Isolation:** The isolation property ensures that transactions that execute concurrently do not affect each other. In the end, 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 a transaction must survive program termination and remain available in the future, even in the presence of system failures.

2.1.2 Concurrency Control

Concurrency control is at the heart of any transaction runtime system. It aims to guarantee the isolation and consistency properties of transactions. This goal is achieved observing concurrently running transactions and identifying *conflicting* operations.

Strict concurrency control, the most basic form of concurrency control, classifies operations according to their access type. The classification contains 3 distinct access types that are named *read*, *write* and *update*. Read operations are observers, meaning that they don't modify the state of a data object. A write operation does alter the state. An update operation corresponds to a read followed by a write. Strict concurrency control makes use of the following conflict table:

Table 2–1: Conflict Table for Strict Concurrency Control

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

The table shows that the only non-conflicting combination is a read-read. This makes sense because read operations do not modify the state of an object and therefore they do not alter the result of other read operations. Other than a read-read, all possible combinations create a conflict.

Using the information of such a conflict table, concurrency control can ensure the isolation property following one of two approaches; *pessimistic* (conservative) or *optimistic* (aggressive). The fundamental difference between these two techniques is the way they deal with conflicts.

Pessimistic Approach. The pessimistic approaches try to *avoid* conflicts by making sure that conflicts never occur in the first place. They achieve this goal by ensuring that every transaction has to get a permission from the concurrency control manager before executing any kind of operation. The concurrency manager then checks to see if the operation would create a possible conflict. If it does, then the transaction is either blocked or aborted.

Lock-based concurrency control is a kind of pessimistic concurrency control that uses locks to implement permissions to perform operations. When invoking an operation on a transactional object, the caller must first request the lock associated with this operation from the concurrency manager of the transactional object. Before granting the lock, the concurrency manager must verify that this new lock does not conflict with any other lock held by other transactions in progress. If the concurrency manager determines that there indeed would be a conflict, the thread requesting the lock is blocked, waiting for the release of the conflicting lock. Otherwise, the lock is granted, and the caller may proceed and execute the operation.

The order in which locks are granted to transactions imposes an execution ordering on the transactions with respect to their conflicting operations. Two-phase locking [EGLT76] ensures serializability by not allowing transactions to acquire any

lock after a lock has been released. This implies in practice that a transaction acquires locks during its execution (1st phase), and releases them at the end once the outcome of the transaction has been determined (2nd phase).

Optimistic Approach. In an optimistic concurrency control scheme [KR81], the approach is to accept that conflicts can occur and then to try and *resolve* them. That means transactions are allowed to perform conflicting operations without being blocked or aborted, but when they attempt to commit, they are validated against a certain criteria. If a transaction is validated, it implies that it has not executed operations that conflict with the operations of other concurrent transactions.

The validation step can be carried out in two ways, backward or forward. Backward validation ensures that the result of a committing transaction has not been invalidated by recently committed transactions. Forward validation ensures that committing transactions do not invalidate the results of the transactions that are still in progress.

2.1.3 Recovery Techniques

It is the responsibility of the recovery support to guarantee atomicity and durability of all transactions at all times. Recovery must make sure that either all modifications made on behalf of a committing transaction are reflected in the state of the accessed objects, or none is, which means that any partial execution of the modifications has been undone. In the event of a system crash, the recovery manager must successfully undo all the modifications made on objects by uncommitted transactions. Furthermore, it must also ensure that the results of transactions that committed before the crash are reflected in the appropriate objects.

There exist two different strategies for performing updates and recovery for transactional objects, namely in-place update and deferred update [KS99].

In-Place Update. This update strategy makes sure that all operations are executed on one main copy of a transactional object. Therefore, the effects of the invocation are instantly made global and they are visible to any concurrent transaction. The undo functionality can be achieved by taking a snapshot (also known as a checkpoint) of the state of a transactional object before it is modified. Later, in the event of a transaction abort or system crash, the states of transactional objects can be conveniently restored using the previously established snapshots.

Deferred Update. When using deferred update, each modifying transaction creates a local copy of the state of a transactional object and executes its operations on this local copy. As a result, changes made by one transaction are not visible to other transactions. The changes are made global upon transaction commit, either by replacing the state of the main copy with that of the local copy or by re-executing the operations on the main copy. Undoing state modifications is trivial with this strategy because it simply means discarding the local copies of a transaction.

2.1.4 Transactional Models

Classical transactions have been developed in the database world, and were intended to perform small units of work that only access a few data items. In time, however, computer scientists wanted to apply transactions to other domains, such as distributed systems or computer assisted design applications. Unfortunately, the classic transaction model is not always well suited in such environments. For instance, as transaction time grows, and the number of data items accesses becomes larger,

the performance of systems based on classical transactions decreases significantly [GR93]. An application with a complex hierarchical structure would greatly benefit from a transactional system that supports fine-grained rollbacks. Alternatively, a model that allows cooperative activities would be beneficial for an application where there is a need for collaborative work.

To address these shortcomings, a number of advanced transaction models have been developed. This section reviews four fundamental transaction models: flat transactions, nested transactions, multithreaded transactions and open multithreaded transactions. For an overview of other advanced transactional models, the interested reader is referred to [Kie03b].

Flat Transactions. Flat (classical) transactions represent the simplest type of transaction. An arbitrary number of statements are encapsulated between a begin transaction statement and an end transaction statement. The ending of a transaction is determined either by a commit statement or an abort statement. This kind of transaction is called flat because every statement inside the transaction is at the same level. Therefore, the transaction will either terminate successfully together with all modifications made to data objects on behalf of the transaction (commit), or it will be rolled back, which means that all changes made to transactional objects will be undone.

Figure 2–1 nicely illustrates the concept of flat transactions. The figure represents the well-known banking example, which performs a transfer of money from one account to another. After the transaction starts, some amount is withdrawn from

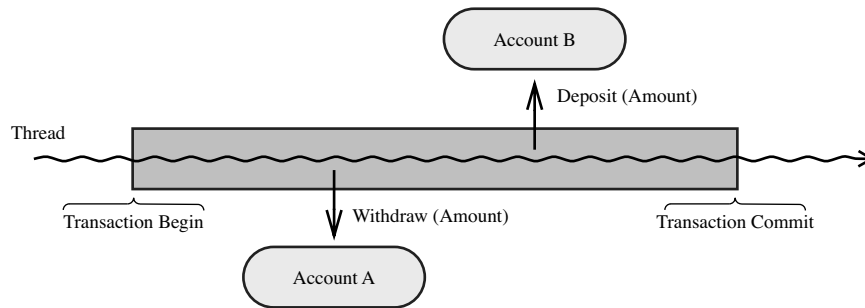


Figure 2-1: Flat Transaction Model

account A and then deposited to account B. If no error occurs during the process, the transaction is committed.

Nested Transactions. In the nested transaction model [Mos81] a transaction is allowed to start subtransactions, implying that it is possible to create a hierarchy of transactions in the form of a tree. In such a transaction tree, a child transaction has access to the data used by its parent transaction, and this rule is recursively applied until reaching the root transaction. Furthermore, when a child transaction commits, its effects are not made globally visible, but instead are delegated to its parent, which is now responsible for committing this data. Aborting a child does not force the abortion of its parents, which gives a more fine-grained control over rollbacks.

The rules for nested transactions are briefly summarized below:

- At any point in time a new transaction can be created. Creating a new transaction inside some other transaction starts a nested transaction.
- The changes on objects are made visible to the outside world only on the commit of the top-level transaction.

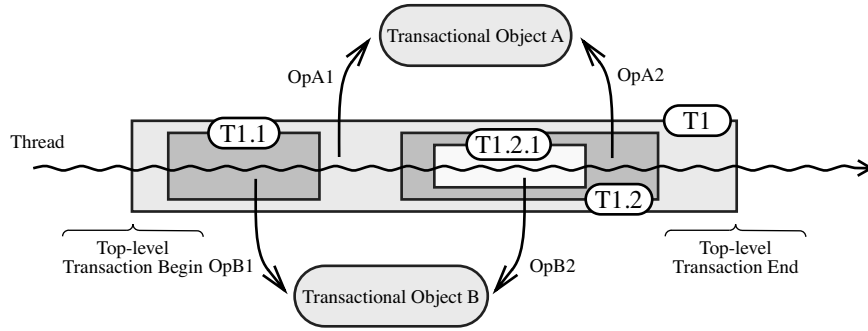


Figure 2-2: Nested Transaction Model

- A parent transaction can only commit once all its subtransactions have committed.
- If a transaction aborts, all its subtransactions are also aborted, independently of their local commit status. This rule is applied recursively down the nesting hierarchy. Therefore, if the top-level transaction is aborted, all its subtransactions are also rolled back.
- Accesses to objects from inside a nested transaction are isolated with respect to the parent transaction, to sibling transactions and to other transactions.

Figure 2-2 shows a top-level transaction T1 with two child transactions T1.1 and T1.2. The second child transaction has yet another child transaction, called transaction T1.2.1. The operation OpB1 is invoked on object B on behalf of the child transaction T1.1. The effects of the operation are made visible to T1 once T1.1 commits. The transaction T1.2.1 that calls OpB2 will be able to see the effects produced by OpB1. Child transactions are not isolated from their parent transaction. As a result, child transaction T1.2 is allowed to call OpA2 although the parent

transaction T1 has previously executed the operation OpA1 on object A. All changes to A and B are made visible to the outside world once T1 commits.

Multithreaded Transactions. Multithreaded Transactions [HKM⁺94] is one of the models that supports cooperation among the threads inside a transaction. In this model, a thread starting a transaction is allowed to spawn additional threads from the inside. Conceptually, the spawning takes place at the transaction border. Before committing or aborting a multithreaded transaction, the spawned threads must run to completion. Threads inside a multithreaded transaction can cooperate with each other, but the model does not control or provide any special means of co-operation. However, they can share transactional objects. The transactional objects are aware of this form of cooperative concurrency.

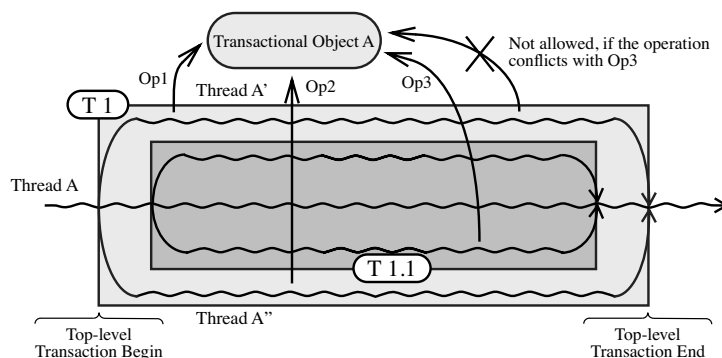


Figure 2–3: Multithreaded Transaction Model

Figure 2–3 shows a multithreaded transaction T1 with a nested multithreaded transaction T1.1. Thread A starts T1, after which it forks 2 additional threads, named Thread A' and Thread A''. Thread A' accesses the transactional object A. Later on, Thread A'' also accesses the transactional object. This is possible since the

threads are both working on behalf of the same transaction. Thread A starts a nested transaction, T1.1, forking again two new threads. One of these threads accesses the transactional object. This is also perfectly legal, since a child transaction may inherit the rights of its parent. There might be a problem later on, if Thread A' tries to access the transactional object again, since child transactions must be run in isolation from their parent. T1.1 commits once all three threads have completed their work. The same rule applies for T1.

Open Multithreaded Transactions. The Open Multithreaded Transaction model [Kie03a] is a transaction model that provides features for controlling and structuring not only accesses to objects, as usual in transaction systems, but also threads taking part in transactions. The model allows several threads to enter the same transaction in order to perform a joint activity. It provides a flexible way of manipulating threads executing inside a transaction by allowing them to be forked and terminated, but it restricts their behavior in order to guarantee correctness of transaction nesting and isolation among transactions.

According to the model, any thread can start a transaction. A thread wishing to work on behalf of an already existing transaction can do so by joining it. In order to join, it has to learn (at run-time) or to know (statically) the identity of the transaction it wishes to join. Threads working on behalf of an open multithreaded transaction are referred to as *participants*. External threads that create or join a transaction are called joined participants; a thread created inside a transaction by some other participant is called a spawned participant.

Within an open multithreaded transaction, threads can access a set of transactional objects. Although individual threads evolve independently inside an open multithreaded transaction, they are allowed to collaborate with other threads of the transaction by accessing the same transactional objects.

All participants finish their work inside a transaction by voting on the transaction outcome. Possible votes are commit and abort. Voting abort is done by raising an external exception. The transaction commits if and only if all participants vote commit.

Once a spawned participant has given its vote, it terminates immediately. Joined participants are not allowed to leave a transaction, i.e. they are blocked, until the outcome of the transaction has been determined. If a participating thread "disappears" from a transaction without voting on its outcome, the transaction is aborted, as this case is treated as an error.

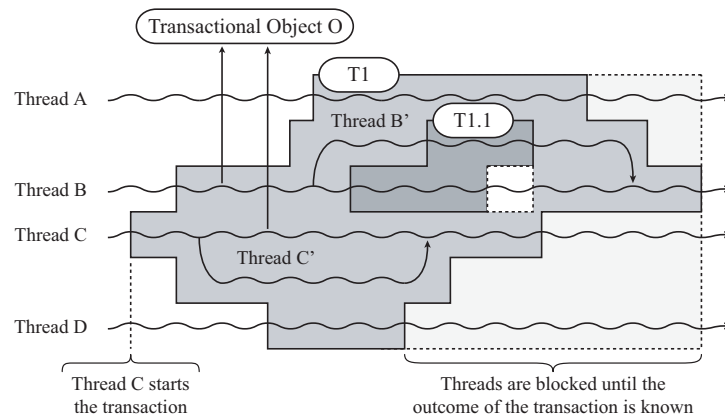


Figure 2-4: Open Multithreaded Transaction Model

Figure 2-4 shows two open multithreaded transactions: T1 and T1.1. Thread C creates the transaction T1, threads A, B and D join it. Threads A, B, C and D are

therefore joined participants of the transaction T1. Inside T1 thread C forks a new thread C' (a spawned participant), which performs some work inside the transaction and then terminates. Thread B also forks a new thread, thread B'. B and B' perform a nested transaction T1.1 inside of T1. B' is a spawned participant of T1, but a joined participant of T1.1. In this example, all participants of T1 vote commit. The joined participants A, C, and D are therefore blocked until the last participant, here thread B, has finished its work and given its vote.

2.2 Aspect-Oriented Programming

Object-orientation is the current mainstream development approach for software systems. When following an object-oriented approach, the real world problems are decomposed into objects that provide an appropriate layer of abstraction, encapsulating related behavior and data in a single unit. This allows developers split complex systems into smaller modules, and hence makes visualization and reasoning about such systems a lot easier. Object-oriented programming (OOP) encourages software reuse by providing language constructs for modularity, encapsulation, inheritance, and polymorphism. Although object-orientation has met great success in modeling and implementing complex software systems, it falls short in capturing behavior that spans across several modules. Attempts to implement such crosscutting behavior in an object-oriented programming language often result in systems that are difficult to reuse or maintain.

Aspect Oriented Programming (AOP) [KLM⁺97] is a promising new technology addressing the shortcomings of object-oriented programming. It introduces new

concepts and constructs that enable the modularization of crosscutting concerns, resulting in systems that are easier to understand, maintain and reuse. AOP is not intended to be a replacement methodology to OOP but a complementary addition. The development of an AOP system typically involves three distinct phases [Lad03]:

Aspectual Decomposition

With AOP, the development process starts by identifying the crosscutting and core concerns. Core concerns are linked to the goals of the system, they encapsulate the core functionality that the system is expected to provide. On the other hand, crosscutting concerns provide extra (secondary) functionality and tend to affect multiple modules of a system. 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 consists of implementing the concerns that are identified in phase one. Aspect-orientation is a concept, so it is not bound to a specific programming language. Basically, there are two distinct approaches for implementing the concerns of an application. An *asymmetric* approach relies on the notion that there is a base body of code that is then augmented with aspects. It permits the independent implementation of the core concerns in either a procedural (such as C) or an object-oriented (such as C++ or Java) platform. Consequently, the crosscutting concerns (i.e., aspects) are typically implemented in an AOP-extension (i.e. AspectJ) of the base language. On the other hand, in a *symmetric* approach there is no distinction between an aspect and a base. All concerns in a system are created equal and can

serve as both aspect and base in different compositions. Most symmetric approaches use programming languages as-is (i.e. Hyper/J), although there are approaches that rely on language extensions as well.

Implementation of the crosscutting concerns requires the understanding of the Join Point Model (JPM). JPM is a very fundamental concept of AOP. It specifies how crosscutting concerns interact with other parts of the application. Specifically, the join point model defines the locations in a program where crosscutting concerns can be applied, a way to select these locations and a means of affecting the behavior at these locations.

Aspectual Recomposition

The rules for re-composing the crosscutting & core concerns are specified in this phase. These rules are typically specified in the aspect, which encapsulates the crosscutting concern, or in a separate configuration or binding module. Both the core concern code and aspects are combined into a final executable form by the aspect weaver. A weaver basically merges all the concerns using the specified weaving rules to produce the final system. As a result, a single aspect can contribute to the implementation of a number of methods, modules, or objects, increasing both reusability and maintainability of the code.

Although it is true that AO promises increased readability, reusability and maintainability, it also requires special attention from the developers. Building an aspect-oriented application is a sophisticated process that involves a lot of effort. Also, since aspect oriented software development is fairly new paradigm it lacks a certain methodology that can be followed. For example, there are various opinions on

what aspect oriented programming can achieve and what it cannot. [KG02] provides a nice discussion that analyzes the limitations of aspect oriented programming. In the paper, the authors try to use AOP techniques to separate concurrency control and failure management concepts from the other parts of the application. However, this attempt shows that concurrency and failure management must be kept in mind throughout the entire application development, that it can not be completely extracted to a separate aspect. In that respect, AOP remains to be insufficient in encapsulating these concerns.

CHAPTER 3

Existing AspectOPTIMA Framework

The thesis aims to extend the existing AspectOPTIMA framework [KG06], by addressing its limitations. Before describing the details of this extension process, it is necessary to have a good understanding of the framework. This chapter briefly describes the existing AspectOPTIMA Framework, which may also be referred as the starting point of the thesis.

3.1 Overview

The motivation for building for such a framework is the observation that although concurrency control and recovery look like two separate concerns at a higher level, they cannot be completely separated at the implementation level. That means, there may be both conflicts and common grounds between these two concerns. One example of a conflict would be that most optimistic concurrency control techniques do not work with in-place update. At the same time, being able to distinguish read from write/update operations is a common functionality that is required by both concurrency control and recovery.

Motivated by this incomplete separation of concerns, the framework applies aspect-oriented design techniques to decompose concurrency control and recovery further, and identifies a set of small aspects, each providing a specific sub-functionality. Using different combinations of these aspects, one can achieve various concurrency

controls and recovery techniques. Section 3.2 describes the aspects that are end results of the decomposition process and section 3.3 presents an example of how these aspects could be combined.

3.2 Decomposition of Aspects

The framework defines ten low-level aspects that each can be applied to an object to provide a well-defined reusable functionality. The aspects are briefly described below:

AccessClassified: The AccessClassified aspect classifies each method that an object defines according to how its execution affects the object's state: each method is classified as a read, write or update method.

Named: The Named aspect associates a name with an object that can be used as a unique means of identification.

Shared: The Shared aspect ensures multiple readers/single writer access to objects – all modifications made to the state of a shared object are performed in mutual exclusion.

Copyable: The Copyable aspect provides functionality to duplicate an object, or replace an object's state with the state from another object.

Serializable: A serializable object knows how to read its state from and write its state to different devices requiring varying data representation formats, e.g. a file or a network connection.

Versioned: A Versioned object can encapsulate multiple copies (versions) of its state. Versions are linked to views, one of which is designated the main view.

A thread can subscribe to a view, and any method call made subsequently by the thread is directed to the associated version.

Tracked: The Tracked aspect provides the functionality to monitor object access in a generic way. It allows a thread to define a region (using begin and end operations) in which object accesses are monitored. At any given time, the thread can obtain all read or modified objects for the current region.

Recoverable: The Recoverable aspect makes it possible to store the state of an object at a given time, and later restore it, if needed. This functionality is sometimes also called "establishing a checkpoint".

AutoRecoverable: The AutoRecoverable aspect provides region-based recovery. It allows a thread to define a region within which recoverable objects are automatically checkpointed before any modifications are made to their state.

Persistent: Persistent objects are objects whose state survives program termination. To achieve this, persistent objects know how to write their state to a non-volatile storage device.

3.3 Recomposition of Aspects

[KG06] further describes that using different combinations of these ten low-level aspects, one can compose several concurrency control and recovery strategies. One example of such a combination is *Pessimistic Lock-Based Concurrency Control with In-Place Update*.

It is implemented in the aspect LockBased, which relies on the fact that all LockBased transactional objects are also AccessClassified, Named, Copyable, Serializable, Shared, Versioned, Tracked, Recoverable, AutoRecoverable and Persistent.

The aspects also assume that the transaction run-time creates a tracked zone, a recoverable zone and a new view when a transaction begins, and ends the zones and the view when a transaction commits or aborts.

Lock-based protocols use locks to implement permissions to perform operations. When a thread invokes an operation on a transactional object on behalf of a transaction, LockBased intercepts the call, forcing the thread to obtain the lock associated with the operation. The kind of lock – read, write or update – is chosen based on the information provided by AccessClassified. Before granting the lock, LockBased verifies that this new lock does not conflict with a lock held by a different transactions in progress. If a conflict is detected, the thread requesting the lock is blocked and has to wait for the release of the conflicting lock. Otherwise, the lock is granted. LockBased then makes sure that in-place update has been selected for this object by calling Recoverable, and allows the call to proceed.

To release all acquired locks when a transaction ends, all transactional objects that are accessed during a transaction have to be monitored. To this end, LockBased depends on Tracked to intercept the call and record the access. Obviously, an object should be tracked only after a lock has been granted.

Next, LockBased depends on AutoRecoverable to intercept the call and to checkpoint the state of the transactional object, if necessary, before it is modified. Since in-place update is being used, Versioned then directs the operation call to the main copy of the object. Finally, Shared intercepts the call and makes sure that no two threads running in the same transaction are modifying the object's state concurrently. After the method has been executed, Shared releases the mutual exclusion

lock. The transactional lock, however, is held until the outcome of the transaction is known.

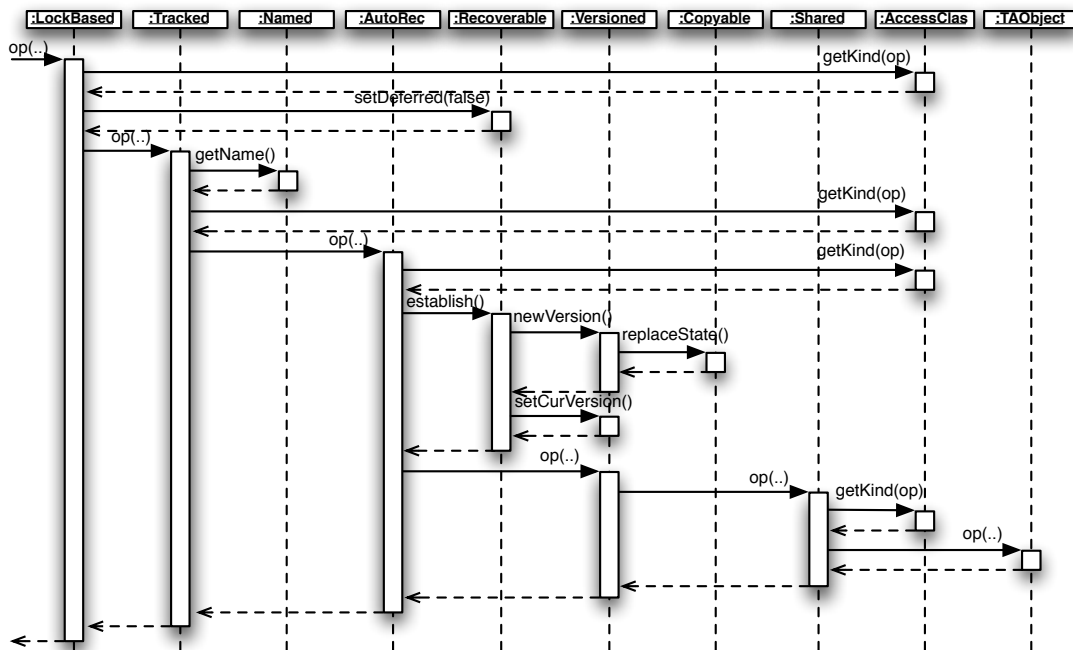


Figure 3–1: Control Flow for LockBased Concurrency Control

Figure 3–1 illustrates this interaction. The sequence diagram depicts how a call to a transactional object is intercepted, and how the individual aspects collaborate to provide the desired functionality.

3.4 Evaluation of the Framework

The framework successfully shows how aspect-oriented techniques can help to decompose concurrency control and recovery into a set of fine-grained aspects. It also presents (section 3.3) how to design various concurrency and recovery techniques using these low-level aspects.

One limitation of the framework is that it does not take into account the concerns that are related to transaction life-time support. The issues such as managing the transaction life-cycle, determining transaction boundaries and handling participant synchronization are beyond the scope of the framework. For example, a careful analysis of section 3.3 reveals that there isn't any definite notification that marks the beginning or the ending of a transaction. Instead, when a thread gets assigned a view and creates the tracked & recoverable zones, it indirectly starts a new transaction. Similarly, when the view and the zones get destructed, the ending of a transaction is implied.

As the authors have pointed out in the paper [KG06], there is a crosscutting concern among the Versioned, Tracked and AutoRecoverable aspects of the framework. Since there is no other construct to mark the transaction boundaries, these three aspects need to define a region of computation (zone / view), in which the threads can be associated with. Creating a region of computation is not a concern of an object, and therefore the authors have chosen not to separate it out.

The removal of the crosscutting concern could only be achieved by integrating the concepts of transaction life-time management into the framework. Performing this integration is a major goal of the thesis and the details of the integration process are provided in the subsequent chapters.

CHAPTER 4

Extending AspectOPTIMA

4.1 Integrating the Transaction Life Cycle

The main goal of this thesis is to integrate transaction life cycle management into the existing AspectOPTIMA framework. Transaction life cycle management includes concerns such as:

- determining the transaction zones/boundaries,
- maintaining the state of the transaction,
- monitoring nesting relationships among transactions,
- controlling participant involvement.

In order to support transaction life cycle management, the old AspectOPTIMA framework had to be expanded and redesigned to include new reusable aspects that encapsulate well-defined concerns of transaction management. With the extended AspectOPTIMA it should be possible to build customized transactional models by combining the aspects provided by the framework in different ways.

As seen in chapter 2, many variations of the classic transaction model have been created to adapt the transaction concept to different application domains. To extend AspectOPTIMA, it is essential to analyze the different concerns of transaction life cycle management to identify common concerns among these models. But even more critical is to understand the essence of a transaction, in order to discover

whether a transaction has its own identity, or whether it is itself an aspect of an object or a thread.

From chapter 2 we know that a transaction is essentially a set of operations executed on a set of data objects by one or multiple threads. Intuitively, since a transaction generally involves multiple data objects, it can not be an aspect of an object. Likewise, since a transaction can be performed by multiple threads concurrently, a transaction is not an aspect of a thread. A transaction is therefore a new concept that exists independently of objects and threads, with its own identity. The reasons are explained in detail in the following subsections.

4.1.1 Why a Transaction is not an Aspect of a Data Object

To begin with, data objects don't play a fundamental role in the life cycle of transactions. An object should not have the responsibility of defining the borders of a transaction. In other words, creating or ending a transaction zone is not a concern of an object. The evaluation of the existing framework shown in chapter 3 supports this argument: the framework does allow the data object aspects to define a region of computation, which vaguely corresponds to the transaction zones. However, this feature creates a crosscutting concern among the aspects Tracked, Versioned and AutoRecoverable of the framework.

Another concern of the transaction lifetime management is being able to keep the state of a transaction. Storing this state with a data object would not be a good idea. Since a transaction can access multiple data objects, the state would have to be replicated in each of the accessed objects. Furthermore, since a data object can take part in several transactions at a given time, the object would have to keep the states

of all the transactions that have accessed it. This would lead to lots of information duplication, and would be highly error prone and inefficient.

4.1.2 Why a Transaction is not an Aspect of a Thread

At first glance, making a transaction an aspect of a thread seems like a reasonable design choice. A transaction does not exist without at least one thread working on its behalf. However, a more in-depth analysis shows that this approach would also cause problems.

First, a transaction may contain more than one thread (participant) working on behalf of it. That means, if keeping the state of a transaction were a concern of the thread, then this state would have to be replicated and properly maintained in each thread that participates in the transaction. This would again be a poorly modularized crosscutting concern. A good solution must allow the threads to share the state of the transaction.

Second, it might be necessary (for recovery purpose, for instance) to keep the state of a transaction even when it is completed, i.e. when all the participant threads have left the transaction. The participants might even continue and start a new transaction, in which case a thread would have to keep storing a list of states of previously completed transactions.

Third, in the case of nested transactions, a thread can actually participate in multiple transactions (the current and the ancestor transactions) at a given time, which again would require storing lists of transaction states with a thread, and hence lead to information duplication and inefficiency.

Finally, depending on the transaction model, it is possible that threads terminate within a transaction, or simply disappear due to some internal faults. Robust transaction implementations should be able to handle this situation by correctly aborting the transaction, and this can only be done if a transaction's lifetime is independent from the life time of its participant threads.

4.2 The Context Idea

The analysis of previous section shows that the thread & object entities remain to be inadequate in capturing the concerns of transaction management. One of the main reasons of this insufficiency is the one-to-many relationship between a transaction & object and transaction & thread. During its lifetime, a transaction may access multiple objects and it may contain many threads (participants). So, what is required is a central unit that can manage these two entities by capturing the many-to-many relationship.

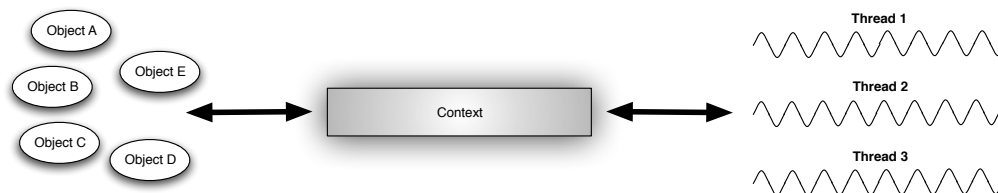


Figure 4–1: Visualization of a Context

The central entity, proposed by the thesis, is called a context (Figure 4–1). A context represents a region of computation with a well-defined state and behavior. It is neither an aspect of an object, nor an aspect of a thread. Instead, a context must be thought as a separate living entity. Similar to an object or a thread, a context

has a life cycle: it gets created, it maintains a state and performs some work during its lifetime and then gets destroyed when it is no longer needed.

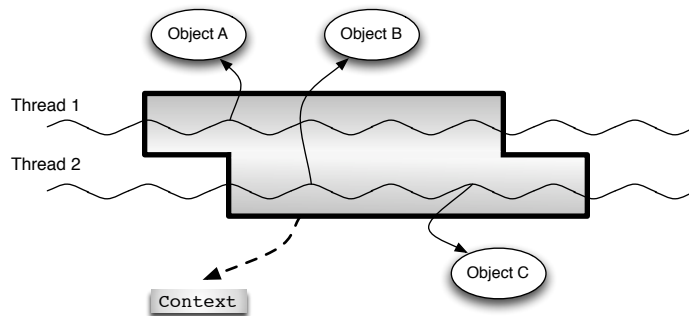


Figure 4–2: Participants performing work on behalf of a Context

In a transaction framework such as AspectOPTIMA, the term context refers to a transactional context. Consequently, the creation of a context corresponds to the beginning of a transaction and destruction of a context implies the ending of a transaction. Therefore, the lifetime of a context defines the boundaries of a transaction.

4.3 Framework with Context

The introduction of the context concept has a great impact on the design of the framework. Now, there are three entities: the object, the context and the thread. For each of these entities, the framework provides well-defined and reusable aspects. **Aspects of the Object:** Using the aspects of the object, one can create a transactional object that can be accessed from a context. The constructed transactional object may support different types of concurrency controls and recovery techniques.

Aspects of the Context: The aspects of a context allow the construction of a variety of transactional models. In coordination with transactional objects that it accesses, the model is free to pick a certain concurrency and recovery scheme.

Aspects of the Thread: The aspects of a thread provide the ability to be a participant of a context, allowing a thread to perform some work on behalf of a context. With this capability, a thread may construct/destroy contexts.

CHAPTER 5

New Design of AspectOPTIMA

This chapter presents the new design of AspectOPTIMA that now takes into account the transaction life cycle management. All the aspects of the framework are described in the next three sections in detail.

Each aspect is identified by a *name*. The name has been carefully chosen to reflect as accurately as possible the functionality that the aspect provides. Certain naming conventions have been followed:

- Aspects ending with "-able" do not modify existing behavior, but rather add state and functionality to the entity they are applied to. These aspects are passive, i.e. their functionality has to be explicitly invoked in order to be executed. *Copyable* described in section 5.1.3 is an example of such an aspect.
- Aspects ending with "-ing" are active, i.e. they automatically perform their functionality at well-defined points in time when the entity they are applied to is accessed. *Checkpointing* described in section 5.3.2 is an example of such an aspect.
- Other aspects, typically ending with "-ed", but sometimes also with "-re" or "-ive", add state to the entity they are applied to, and make sure that this state is correctly initialized / accessed. *Versioned* described in section 5.1.6 is an example of such an aspect.

Each aspect description in the following sections contains a motivation section, a section describing the functionality of the aspect and its dependencies on other aspects, and a reusability section that describes potential uses of the aspect out of the context of the AspectOPTIMA framework.

5.1 Aspects of Object

The integration of the transaction support management has caused certain changes in the aspects of an object. There were some new aspects added: Lockable, ContextAware and ContextTracing. There were minor modifications on the design of the following aspects: Shared, Versioned and Persistable. The aspects Traceable and Checkpointable provide some part of the functionality that Tracked and Recoverable used to provide in the old framework. The aspects AccessClassified, Named, Copyable and Serializable are exactly the same as in the previous framework.

The following figure displays all the aspects of an object and also captures the dependencies between these aspects. The aspects that have to intercept calls to objects are stereotyped `<i>` (for interceptors).

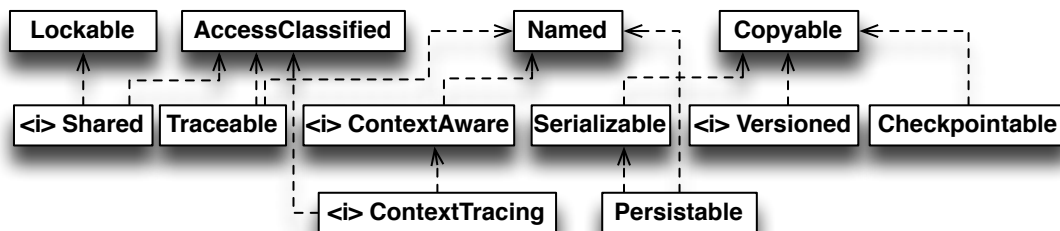


Figure 5–1: Aspects of Object

5.1.1 Named

Motivation: One of the most essential properties of an object is its identity. It makes an object unique by making sure that it is distinguishable from all other objects. The use of a reference pointing to the memory location of an object is a common practice for uniquely identifying an object. However, the lifetime of transactional object is not bounded by the lifetime of an application. Since the state of a transactional object survives program termination, there must be a more persistent way, which can identify a transactional object that remains valid during several executions of the same program.

The unique identification of a transactional object is also helpful when working with different concurrency & recovery controls. Depending on the chosen concurrency control and recovery scheme, there might exist multiple copies of the state of a transactional object. Although, there may exist multiple copies, these copies in fact represent one application object, which is uniquely identified by the system.

Formerly, it has been shown [KRS00] that an object name in the form of a string is an elegant solution for uniform object identification. The functionality of associating a unique name with an object is provided by the aspect *Named*. At creation time, the object is given a name that is associated with its identify, and this name remains valid throughout the its entire lifetime. It should be possible to obtain the name of a given object and to retrieve an object given its name.

Functionality:

- All creator operations must associate a unique name with the object, as soon as it gets created.

- *Object* `getObject(String s)` and `String getName(Object o)` operations provide the mapping from an object to its name and vice versa.

Reusability: Named could be used as a standalone aspect whenever there is a need to refer to an object by its name.

5.1.2 AccessClassified

Motivation: In order to perform concurrency control & recovery more efficiently, one has to classify the operations on transactional objects. The classification of these operations could be done according to how they affect the state of an object: read operations that only observe the state of an object, write and update operations, which read and write the state of an object. The AccessClassified aspect encapsulates this classification.

Functionality:

- Every method of the object is classified as either a read, write or an update operation. Given a method identifier, `getAccessType(Method m)` returns the access type (i.e. read, write or update) of the provided method.

Reusability: The classification provided by AccessClassified is very helpful in many contexts. It could be used in multi-processor systems to implement intelligent caching strategies, or help to choose among different communication algorithms and replication strategies in distributed systems.

5.1.3 Copyable

Motivation: An object encapsulates some state. This state is initialized at the creation time of an object, and it can be modified by each method invocation. Certain concurrency controls and recovery schemes (such as deferred update) might want to

duplicate the state of an object. Alternatively, it may be necessary to replace the state of an object with state of another object of the same class. The Copyable aspect provides these functionalities.

Functionality:

- The method *Object clone()* creates an identical copy of the object.
- *replaceState(State source)* could be used to replace the state of an object, with the state of the provided source object.

There are two ways to duplicate the state of an object: using deep copy and shallow copy. When an object A stores in its state a reference to an object B, deep copy also clones / replaces the state of B when cloning / replacing the state of A. Recursively, if B refers to other objects, they are cloned / replaced as well. Shallow copy, on the other hand, only clones / replaces the state of A. A flexible Copyable aspect implementation should support both of these techniques because different applications may prefer different techniques.

Reusability: Copyable is used whenever an object's state must be duplicated or copied into another object of the same class. As an example, replication systems can make use of the Copyable aspect since they frequently perform cloning operations. It is so widely used that most programming languages provide the functionality of Copyable (e.g. the Java Object clone() method that can be invoked on all objects that implement the Cloneable interface).

5.1.4 Lockable

Motivation: The use of locks is a well-known technique to achieve pessimistic concurrency control. In this concurrency scheme, a transaction has to get permission

from the concurrency management before performing any operation on the objects. One way to implement the permission step is to associate a lock with each transactional object and make sure that the transactions acquire this lock before accessing an object. The Lockable aspect provides the functionality of attaching locks to an object, and defines an interface for acquiring and releasing locks. The locks provided by lockable distinguish read and write operations, and hence implement the multiple readers // single writer paradigm.

Locking objects can be useful in many situations, and sometimes it might even make sense to associate several locks of a different type with the same object. In order to support that, Lockable provides an interface for the creation of new lock types. Locks of different type, however, are independent and do not interfere with each other. It is the responsibility of the users of Lockable to ensure that, if there is an application-level dependency between locks, the locks are used in a consistent way.

Functionality:

- Methods *acquireReadLock(String type)* and *acquireWriteLock(String type)* acquire the lock of type *type* of an object. If this is the first time that a lock of this type is requested, a new lock instance is created and linked to the type. Otherwise, the acquire operation is performed on the already existing lock of that type.
- Similarly, *releaseReadLock(String type)* and *releaseWriteLock(String type)* release the lock of type *type* of an object.

Reusability: The Lockable aspect could be used in any setting where there is a need for mutual exclusion. It could be used in the implementation of various concurrency patterns, such as a synchronized block or a reader/writer lock.

5.1.5 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. Therefore, it is necessary to prevent the threads, which jointly work on behalf 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 modifier (write or update) operation or multiple concurrent observer operations (read) — assuming no modification operation is in progress.

Functionality:

- Before executing the method body of a shared object, a read or a write lock is acquired.
- When returning from the method, the previously acquired lock is released.

Shared depends on two aspects in order to perform its functionality. It uses AccessClassified to distinguish between observer and modifier operations. For providing mutual exclusion through the use of locks, Shared makes use of the Lockable aspect.

Reusability: Shared can be used as a standalone aspect in any concurrent application to guarantee data consistency of shared objects.

5.1.6 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. One way of achieving this isolation property is to ensure that each transaction has its own view of the state of transactional objects. A thread within a transaction should only see the updates made by other participants of the same transaction, but not the updates made by other transactions. The concurrency & recovery schemes, which use this isolation technique, have to create multiple copies of the state of a transactional object in order to isolate state changes made by different transactions. This functionality is provided by the Versioned aspect.

A Versioned object encapsulates several versions of the state of a transactional object, with one version designated as the main version. A thread is allowed to register itself to a specific version of a Versioned object. In other words, whenever a thread invokes a method of the Versioned object, the actual method invocation is performed on the version that the thread had previously registered to. If the thread is not registered to any version, then the call is forwarded to the main version.

Functionality:

- *Version newVersion()* creates a new version. The returned Version object is a handle to the newly created version.
- *deleteVersion(Version v)* deletes the version v.
- *setMainVersionAs(Version v)* designates the main version of the object.
- *setVersionID(Version v)* and *Version getVersionID()* manipulates / observes the version identifiers of the object.

- *registerVersion(Version v)* allows a thread to register itself to the version *v*. Similarly, *unregisterVersion(Version v)* cancels the registration.
- Any method invocations on the transactional object are directed either to the version registered to the calling thread, or else by default to the main version.

The Versioned aspect relies on the presence of the Copyable aspect to duplicate the object's state when a new version is created.

Reusability: Versioned can be used as a standalone aspect in any application that needs multiple views of an object's state. Using the functionality provided by the aspect, the application can manipulate each version of an object independently.

5.1.7 Checkpointable

Motivation: Whenever a transaction aborts, there is a need to undo all the changes that the transaction has made on the transactional objects. Transactions can achieve this by taking a snapshot (establishing a checkpoint) of the objects before changing them. Upon transaction abort, these previously stored states can simply be restored in order to undo the changes. These operations however require some assistance from the transactional objects. The objects must be able to create a checkpoint, and restore the checkpoints upon request. The Checkpointable aspect provides this functionality.

Functionality:

- *establishCheckpoint()* establishes a checkpoint of the state of the object.
- In order to restore the most recently established checkpoint, the method *restoreCheckpoint()* is used.
- *discardCheckpoint()* method discards the most recently established checkpoint.

The Checkpointable aspect depends on the Copyable aspect to duplicate the object's state when a new checkpoint is established, and to replace its state when a checkpoint is restored.

Reusability: The Checkpointable aspect can be used in any application, where there is a need for the undo (rollback) functionality.

5.1.8 Traceable

Motivation: In any context that performs work, there may be a need to trace the objects that have been accessed, and the kind of access that has taken place. The tracing operation requires some assistance from the objects; the objects must provide the information to be remembered. In a transactional context, this information is usually the identity of an object and the type of interaction between a context and an object. Traceable objects can present this information by semantically combining the information provided by Named and AccessClassified aspects.

Functionality:

- The Traceable aspect allows an object to create a trace of itself using the *createMyTrace(Method m)* method. The result of this method is a data structure that encapsulates useful information about the object and current operation: in our case it contains the name of the object and the access type of the invoked method.

The Traceable aspect relies on the Named aspect to uniquely identify an object, and relies on AccessClassified to classify the type of interaction as *read*, *write* or *update*.

Reusability: The Traceable aspect can be used in any context where there is a need to remember objects and operations or any other meta-data about objects that have been used within a certain context. The information provided by the aspect can, for instance, be useful for logging and debugging purpose.

5.1.9 Serializable

Motivation: The state of a transactional object does not necessarily reside in main memory forever. Since a transactional object survives program termination, it may be necessary to move an object's state to a different location, e.g. to a file, a database or over the network to a different machine. The main memory representation of the object must therefore be transformed to the appropriate representation of the destination location.

Serializable provides this functionality. A serializable object knows how to read its state from and write its state to backends requiring varying data representation formats. It is an incarnation of the Serializer pattern described in [RSB⁺98].

Functionality:

- *readFrom(Reader reader)* restores the state of the object by reading it from a backend.
- *createFrom(Reader reader)* creates a new object and initializes it with the state read from a backend.
- *writeTo(Writer writer)* saves the state of the object to a backend.

Serializable relies on Copyable to replace the state of an object with that of a previously serialized object of the same class. Just like with Copyable, serialization

can be deep or shallow. Again, the ideal way of performing serialization is application dependent. Serializable should therefore be customizable to specific application needs.

Reusability: Serializable can be used in many situations, e.g., for writing an object's state to a file, sending the state over the network, or displaying an object's state. Serialization is so handy that modern programming languages usually provide a default serialization implementation for objects. The default serialization can usually be overridden with customized serialization, if needed.

5.1.10 Persistable

Motivation: Transactional objects are objects whose state survives program termination. In order to achieve this goal, the objects should know how to write their state to stable storage [LS79], i.e. a reliable secondary storage such as a mirrored hard disk or a database. Consequently, it must be possible to reinitialize the state of an object based on the content of the storage device.

Functionality:

- All creator operations (i.e. constructors) of an object must associate a well-defined location on a storage device with the object.
- Operations to load/save the state of the object from/to the associated storage device are provided.
- An explicit operation to destroy a persistable object is needed as well, or else an implicit garbage collector that identifies and destroys unreachable persistent objects [FS96]. When the object ceases to exist, the space on the associated storage device has to be freed as well.

Persistable aspect relies on the Serializable aspect to transform the object's state into a flat stream of bytes. Furthermore, it needs the Named aspect assuming that the name of an object would help to designate a valid location on the secondary storage device.

Reusability: Persistable could be used in any application where an object's state has to survive program termination.

5.1.11 ContextAware

Motivation: Objects that are accessed from within a context have to obey a set of rules. The rules depend on the kind of context, and maybe even on its state. Therefore, an object that is accessed by a thread that is part of a context should notify the context of the upcoming access. That way, the context can take actions to implement the specified rules, if necessary.

The ContextAware aspect performs this functionality, and hence makes the object to which it is applied eligible for taking part in a context.

Functionality:

- Whenever a method of a ContextAware object is invoked, the aspect intercepts it and examines the origin of the call. If the call comes from a thread that participates in a context, then the object notifies that context about the method invocation.

Each accessed object should be uniquely identifiable in a context. Therefore, ContextAware depends on the Named aspect to provide an identity for the object.

Reusability: Whenever an operation invocation on an application object is part of a bigger computation that has to follow certain rules, the ContextAware aspect can be used.

5.1.12 ContextTracing

Motivation: Certain recovery and concurrency control schemes, in particular optimistic ones, have to compare the data they modified with the data that was read by other concurrently running transactions before they commit. This task can be facilitated if every object keeps a list of all the transactions that have accessed it. The ContextTracing aspect provides this functionality by keeping a list of all contexts that have interacted with the object.

Functionality:

- *List* `getAllContexts()` retrieves all the contexts that have accessed the object.
- Whenever a context accesses a ContextTracing object by invoking one of its methods, the aspect intercepts this call and adds the context to its list.

The aspect relies on the ContextAware to identify the origin of the call. ContextTracing also needs to store the type of interaction (read/write/update) for each method invocation made by the context. Therefore, ContextTracing also needs the presence of the AccessClassified aspect.

Reusability: In the presence of a context, the ContextTracing aspect can perform logging operations. For example, a ContextTracing object is able to trace all threads that have worked with the object.

5.2 Aspects of Thread

The integration of transaction management concepts into the framework lead to the creation of new aspects that apply to threads. The figure (Figure 5–2) displays these aspects and also mark the aspects that have to intercept calls with a `<i>` stereotype. This section is dedicated to the description of the aspects of the thread entity.



Figure 5–2: Aspects of Thread

5.2.1 ContextParticipant

Motivation: Threads that work on behalf of a transaction are called *participants*. Participants create the transaction, perform some work and then leave the transaction.

The `ContextParticipant` aspect allows a thread to be a participant of a context. In other words, the aspect connects a thread to a context by constructing the link between the two.

Functionality:

- Methods `Context getContext()` and `setContext(Context c)` monitor and manipulate a participant's relationship with a context.
- `ContextParticipant` threads have the ability to *create* new contexts and *leave* contexts when they have completed their work.

Reusability: In any application that structures its execution by means of contexts (be they transactional or of a different kind), the ContextParticipant aspect can be used to make threads eligible to participate in a context.

5.2.2 Collaborating

Motivation: In certain transactional models, such as Open Multithreaded Transactions, multiple threads may be working on behalf of the same transaction. In those kind of contexts, threads work collaboratively toward a particular goal. The life-cycle of a collaborative transaction is slightly different from an individual one. The number of participants in a collaborative context could be fixed in advance, but does not have to be. Collaborative contexts allow new participants to join during the execution of a context. The Collaborating aspect encapsulates this functionality.

Functionality:

- The Collaborating aspect provides a *join* operation that allows a thread to join an already existing context.
- Whenever a collaborative participant creates a new thread inside the boundaries of a context, the Collaborating aspect makes sure that the newly created thread automatically becomes a participant of the enclosing context. In other words, Collaborating aspects allow the spawning of new participants.
- A Collaborating thread is allowed to perform a *close* operation, which subsequently forbids the joining of additional participants.

The Collaborating aspect relies on the ContextParticipant aspect to provide the link between a thread and a context. In other words, a thread has to first be able participate in an individual context, in order to take part in a collaborative context.

Reusability: In any application where there is a need for threads to perform collaborative work or synchronize, the Collaborating aspect can be helpful.

5.2.3 OutcomeAffecting

Motivation: Generally, work performed as part of a context is done with a certain goal in mind. Not always, however, can this goal be reached successfully. To capture this, a context can have an outcome (such as successful or unsuccessful). Depending on the context, this outcome is affected by various factors. One of these factors is the opinion of the entities that have worked on behalf of this context, since they performed the actual work. The OutcomeAffecting aspect enables the participants to submit their opinions about the outcome of the context that they have worked for.

Functionality:

- An OutcomeAffecting thread has the ability of contributing to the outcome of a context.
- Whenever an OutcomeAffecting thread leaves a context without presenting its decision on the outcome, the aspect automatically submits the default opinion.

The OutcomeAffecting aspect depends on the ContextParticipant aspect to provide the basic functionality of being able to participate in a context.

Reusability: The OutcomeAffecting aspect can be used whenever there is a need to make a decision that involves multiple participants.

5.3 Aspects of Context

The context entity represents a well-defined region of computation, in which threads may perform some work by accessing objects. The design of the framework

requires that threads participating in a context are ContextParticipant threads. Similarly, a context can only perform work on ContextAware objects.

Using the aspects of the framework, it is possible to customize the behavior of a context. Each of the following aspects adds properties to the context by adding some functionality, introducing some state or controlling the involvement of its participants according to certain rules. The figure 5–3 summarizes all the aspects of a context and also displays the dependency relationships between the aspects. The aspects that have to intercept calls are stereotyped <i>.

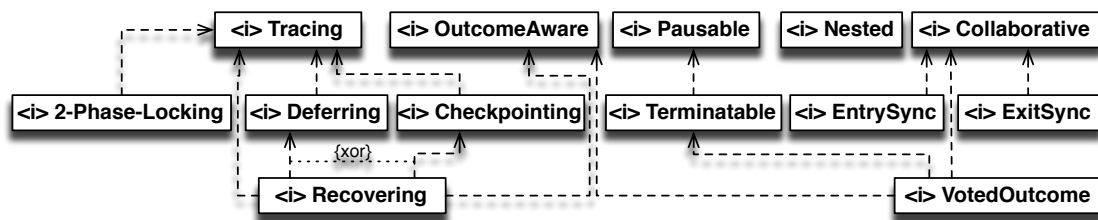


Figure 5–3: Aspects of Context

5.3.1 Tracing

Motivation: One of the most common functionality that is required by any concurrency control and recovery scheme is keeping track of all transactional objects that are accessed from within a context. In addition, the transaction runtime must also record the type of access (i.e. read, write or update) for each accessed object. This information is, for instance, used by the recovery manager to identify the modified objects, so that their states can be rolled back in case of an abort. Similarly, an optimistic concurrency manager needs that information when checking for conflicts between transactions.

The Tracing context provides this functionality by storing a list of all the objects that it has accessed with the context. Some cooperation is required from the object side; it must be *Traceable*, i.e. it provides the information that is to be stored. In our case this information comprises the object's identity and the access type of the invoked method.

Functionality:

- Whenever a context accesses a transactional object, the Tracing aspect intercepts the call and checks to see if the object is *Traceable*. If the object is *Traceable*, the Tracing aspect asks the *Traceable* object to create its trace and stores the created trace with the context.
- The *List getModifiedObjects()* method returns the set of objects that have been modified by the context.
- All the objects that the context has read (observed the state of) can be retrieved using the *List getReadObjects()* method.

Reusability: A Tracing context can be used in a standalone way to monitor object access made by arbitrary pieces of code, for instance for the sake of logging and debugging.

5.3.2 Checkpointing

Motivation: The transaction runtime must be able to undo state changes made on behalf of a transaction when it aborts. One way to provide this functionality is through taking a snapshot (checkpoint) of the object before modifying it. These established checkpoints can later be used to undo all state changes made on behalf of a transaction.

Functionality:

- Whenever a Checkpointable object is modified for the first time by the context, a checkpoint of the object is automatically established.
- When the context terminates, all the checkpoints that the context has previously established are discarded.

The Checkpointing aspect relies on the Tracing aspect to keep the traces of all the accessed objects. This functionality is necessary when determining whether an object is being modified for the first time. Furthermore, a Checkpointing context requires the accessed objects to be Checkpointable.

Reusability: Checkpointing can be used in any application that wants to be able to recover state changes made to a set of objects by a set of operations. For instance, it can be used to implement simple undo functionality.

5.3.3 Deferring

Motivation: In certain recovery and concurrency control schemes, there is a need for each transaction to have its own view of the objects that it accesses. In other words, transactions need to execute in complete isolation from each other. This is possible if each transaction gets to work on its own separate copy of the state of a transactional object. Before modifying any object, the transaction duplicates the state of the object and performs the modification on the duplicate. Thus, the actual update operation on the object is deferred. Only when the context ends, the modifications are performed by replacing the state of the main copy with the state of the local copy.

Functionality:

- Whenever a Versioned object is modified for the first time by the context, the Deferring context creates a new version of the object and registers itself to this newly created version. As a result, the subsequent calls made by the context are forwarded to this specific version.
- When the context terminates, the state of the main version of each modified object is replaced by the state of the local copy (a functionality provided by the Versioned aspect). As a result, all the changes that the context has performed are now globally visible.

In order to find out whether an object was previously modified or not, the Deferring aspect relies on the Tracing aspect. In addition, the accessed objects have to be Versioned, so that the Deferring context can duplicate their states and work on its own versions.

Reusability: The Deferring aspect can be used in any application when a unit of work has to be executed in total isolation from the rest of the application.

5.3.4 Recovering

Motivation: When a transaction aborts, the transaction run-time support must be able to undo all state modifications made on the transactional objects on behalf of the transaction. This activity is also called rollback. The recovery management is responsible for ensuring the all-or-nothing property of transactions by providing this rollback functionality in case of an abort.

The AspectOPTIMA framework supports two types of recovery depending on the update strategy adopted by a transaction. In the in-place update approach, all operations are executed on one main copy of a transactional object. In the deferred

update technique, each modifying transaction creates a local copy of a transactional object and executes its operations on this local copy. The actions taken by the recovery manager depend on the selected update strategy. Integration of other types of recovery into the framework, such as logical recovery, is left for future work (see chapter 10).

Functionality:

- The Recovering aspect automatically undoes the state changes made on behalf of a context when the context terminates unsuccessfully. In order to accomplish this, the Recovering aspect checks the outcome of the context when the last participant has left. If the outcome of the context is negative (i.e. abort decision), the Recovering aspect first retrieves the set of objects that were modified by the context. Then it starts the recovery process:

In-Place Update: For each modified object, the previously established checkpoint is restored.

Deferred Update: With this approach, the context has previously created a local copy (version) for each of the modified objects. The recovery process, in this case, simply consists of discarding these local versions.

The decision of whether to perform recovery or not depends on the outcome of the context. Therefore, the Recovering context relies on the OutcomeAware aspect to attach a specific outcome to a context. In order to rollback effects of a transaction, the Recovering aspect needs the set of objects that were modified by the transaction. Hence, the Recovering aspect depends on the Tracing aspect. Furthermore, the Recovering aspect also needs one of the two aspects that provides the basis for

recovery: Checkpointing or Deferring. The Recovering aspect either depends on the Checkpointing aspect to establish a checkpoint of each object before modifying it, or else relies on the Deferring aspect to ensure that the context creates and works on a local copy for each modified object.

Reusability: The Recovering aspect can be used in any application that wants to automatically recover state changes made by an unsuccessful unit of work.

5.3.5 Two-Phase Locking

Motivation: Concurrency control is a fundamental part of any transaction run-time system because it guarantees the isolation property. One way to achieve this isolation property is to use the two-phase locking [EGLT76] strategy. By imposing an execution order on concurrently executing transactions when they access transactional objects, two-phase locking ensures the serializability property for the operation traces generated by these transactions. The two-phase locking strategy divides a transaction execution into two phases: a growing phase, in which locks are acquired before a operation on an object is executed, and a shrinking phase (once the outcome of the transaction is known), in which all locks are released.

Functionality:

- Before accessing any transactional object, Two-Phase Locking checks to see whether the object has been accessed before by the context. If this is the first time that the context interacts with the object, the aspect makes sure that the context acquires the transactional lock that is associated with the object.
- When the context ends, the Two-Phase Locking aspect releases all the acquired locks.

A Two-Phase Locking context acquires the transactional locks of the objects that it accesses. In order to do that, every object must possess a lock that can be acquired by the context — it must be Lockable. Furthermore, the context must keep traces of the accessed objects so that it can acquire the transactional lock of an object when it is accessed for the first time. This functionality is provided by the Tracing aspect.

Reusability: The Two-Phase Locking aspect can be used to isolate threads participating in different contexts.

5.3.6 Nested

Motivation: Nesting is a fundamental idea that appears in various fields of computer science. For example, nesting is found at the heart of recursion, which is one of the most well known concepts in programming. In the transaction world, the ideas introduced by nesting form the basis of an essential transactional model: nested transactions. A nested transaction occurs when a new transaction is started by an instruction that is already inside an existing transaction. These nested transactions (subtransactions) help in achieving a more fine-grained rollback structure and thus allows complex computations to be split into smaller pieces.

Functionality:

- The Nested aspect establishes the nesting links between parent and child contexts, i.e. it links a context to a single parent, if any, and many children, if any.
- Whenever a participant leaves a nested context (sub-context), the aspect makes sure that the participant is (re-)associated with the parent context.

Reusability: The Nested aspect is helpful in any application where units of work, represented by the context concept, form a hierarchy.

5.3.7 OutcomeAware

Motivation: In general, transactions are performed with a specific goal in mind, and most of the time the transaction participants want the transaction to commit. A voluntary abort is only performed when for some reason the goal could not be reached. If the transaction commits, the effects of the transaction are made permanent and visible to others, i.e. the transaction was successful and hence the goal was reached. In case of an abort, all the changes of the transaction are undone, i.e. the transaction was unsuccessful, and the goal was not reached (but, thanks to atomicity, the system is still in a consistent state). The functionality of the OutcomeAware aspect is to attaching the notion of outcome to a context. The outcome of a context can either be successful or unsuccessful.

Functionality:

- The OutcomeAware aspect provides a binary outcome for a context and allows this outcome to be observed / modified.

Reusability: The OutcomeAware aspect can be used to associate an outcome to any kind of context, not only transactional ones.

5.3.8 Collaborative

Motivation: Atomic actions [CR86] [LA90] is a fault tolerance model that allows a fixed number of participants to enter an action and cooperate inside it to achieve joint goals. These participants share work and explicitly exchange information in order to complete the action successfully. Similarly, there are some transactional models,

such as Open Multithreaded Transactions [Kie03a], that allow multiple threads to work on behalf of a single transaction. These transactional models are constructed by combining the ideas of competitive and collaborative environments.

The common point in all these models is the cooperation of multiple threads for achieving a specific goal. The Collaborative aspect ensures that a context can support collaborative work by managing multiple participants.

Functionality:

- The aspect ensures that the participants can dynamically join a Collaborative context.
- The Collaborative aspect keeps a list of participants that are working on behalf of the context. It distinguishes between the participants that have joined the context (joined participants), and the participants that were created inside the context (spawned participants).
- After calling the method *closeContext()* on a Collaborative context, other participants can no longer join the context.
- The method *setMaximumParticipantCount(int value)* optionally determines the maximum number of participants that can work on behalf of the context.

Once this number is reached, Collaborative closes the context automatically.

Reusability: The Collaborative aspect can be used separately to perform any kind of work which requires the cooperation of multiple entities (threads, processes).

5.3.9 EntrySynchronizing

Motivation: In a collaborative environment, multiple entities work together in order to achieve a specific goal. This shared goal might only be achieved if every participant is present and ready to perform its role. The EntrySynchronizing aspect, if activated, waits for the presence of a certain number of participants before allowing any participant to start working within a context.

Functionality:

- The method *setMinimumNumberOfParticipants(int value)* specifies the number of participants that the context needs in order to start performing work. The default minimum number is 1.
- Whenever a participant joins a context and a minimum number of participants has been previously specified, the EntrySynchronizing aspect checks to see if total number of participants is equal or greater than the specified minimum count. If the minimum number is not met yet, then the participant is forced to wait until there are enough participants.

EntrySynchronizing requires the presence of the Collaborative aspect to support multiple participants. In individual contexts there is no need for synchronization.

Reusability: The EntrySynchronizing aspect can be used in any application where there is a need to synchronize multiple threads before allowing them to start performing a task.

5.3.10 ExitSynchronizing

Motivation: In transactional models that support multithreading, such as Open Multithreaded Transactions, the participants of a transaction can not be allowed

to leave when they finished their work, but have to be blocked until the outcome of the transaction is known. This is essential to guarantee proper isolation among transactions. The ExitSynchronizing aspect provides this functionality.

Functionality:

- Whenever a participant tries to leave the context, it is blocked by the ExitSynchronizing aspect. Only when the last participant of the context leaves, all other waiting participants are released.

The functionality presented by the ExitSynchronizing aspect only make sense if a context can provide support for multiple participants. Therefore, ExitSynchronizing aspect depends on the Collaborative aspect.

Reusability: The ExitSynchronizing aspect can be used in any application where there is a need to synchronize multiple threads after having performed a task.

5.3.11 Pausable

Motivation: A context may encounter various errors or exceptional situations during its lifetime. In order to cope with these errors, the context may need to stop working for a period time, and try to fix the errors before it continues working again. That means, a context first needs to tell its participants to pause and later ask them to continue when the context is ready to perform work once again. The atomic actions model [LA90] uses a similar idea as a part of its exception resolution strategy. Whenever a participant raises an exception, there is a need to pause the other participants in order to check for the occurrence of multiple concurrent exceptions. When the raised exceptions are resolved, all the participants are notified of the resolution and they may continue working on behalf of the context.

Functionality:

- A Pausable context communicates with its participants in order to suspend their work temporarily when the method `pauseContext()` is invoked.
- When the `resumeContext()` method is invoked, the context instructs the participants to continue performing their work.

Reusability: The Pausable aspect is helpful in any application where there is a need to suspend ongoing work for a period of time.

5.3.12 Terminatable

Motivation: When it is decided that a transaction must abort, all participants that work on behalf of the transaction should be informed about this decision as soon as possible. After the decision is made, it does not make sense to perform any further work because the transaction management anyhow undoes all the changes made on behalf of the transaction. In such a situation there is a need for a mechanism to stop the participants working on behalf of a transaction and inform them of the abort. The Terminatable aspect provides this functionality.

Functionality:

- The method `terminateContext()` stops the execution of a context and removes all of its participants.

Terminatable relies on the presence of Pausable to interrupt and get the attention of all the participants of a transaction.

Reusability: The Terminatable aspect is usable whenever there is a need to stop to perform work within a context. For example, a converging computation that is

performed within a Terminatable context could be terminated prematurely if the current approximation is sufficiently accurate.

5.3.13 OutcomeVoted

Motivation: Certain transactional models allow multiple participants to work on behalf of the same transaction. Since the work is performed in collaboration, the decision of whether a transaction commits or aborts should also be made in a collaborative fashion. Such a collaborative decision process is called voting. For instance, in Open Multithreaded Transactions all participants finish their work inside a transaction by voting on the transaction outcome. The OutcomeVoted aspect provides the functionality to decide on the outcome of a context by means of a voting process.

Functionality:

- The method *voteForContext(Vote)* is provided to participants to submit their vote.
- The outcome of an OutcomeVoted context is determined by the votes of its participants, based on a user-defined voting strategy. In the default scheme, a positive outcome could be achieved if and only if all the participants vote positively.
- If a decision is reached before every participant has given his vote, then the aspect terminates the context and notifies the participants about the outcome.

The OutcomeVoted aspect provides a mechanism to decide the outcome of a context, and hence depends on OutcomeAware. Since voting only makes sense when multiple participants are involved, OutcomeVoted relies on the Collaborative aspect to make sure that a context supports multiple participants. Some voting schemes

allow a decision to be made before all participants have given their vote. To support this situation, OutcomeVoted requires the presence of the Terminatable aspect in order to terminate a context after a decision has been reached.

Reusability: The OutcomeVoted aspect can be used whenever a decision process which involves multiple members has to be conducted.

CHAPTER 6

Using AspectOPTIMA: Reconstructing Transaction Models from Aspects

This chapter shows how the individual aspects presented in the previous chapter can be combined in many different ways to create various transaction models using different concurrency control and recovery techniques. The first subsection, however, demonstrates that the aspects can also be used to perform non-transactional activities.

6.1 Performance Monitoring Context

GOAL: The goal is to create a context that measures the performance of an application by timing individual method invocations. Such a context can, for instance, serve as a benchmark for analyzing how well an application performs in a concurrent environment. In order to achieve a detailed analysis, the context classifies the invoked methods according to their access type (i.e. read/write/update) and provides statistics for each type. The context can contain any number of threads, which execute concurrently and randomly access the data objects. In order to measure the overhead induced by concurrency correctly, the threads that work in this context should begin performing their operations at the same time.

DESIGN: The table below shows the aspects that were used to construct the performance-monitoring context.

Table 6–1: The Design of Performance Monitoring Example

Aspects		
<i>Object</i>	<i>Context</i>	<i>Thread</i>
Shared Traceable ContextAware	Tracing Collaborative EntrySynchronizing PerformanceMonitoring	Collaborating

Since the threads are executing their operations concurrently on the same set of objects, there is a need for the **Shared** aspect to preserve data consistency of the objects. The objects also have to be **Traceable**, so that the context can keep traces of the objects that were accessed by the context. Furthermore, the Traceable aspect is also required to distinguish between different types of methods (i.e. read/write/update). Finally, the object has to be **ContextAware** so that it becomes eligible to be accessed by the context.

The threads have to be **Collaborating** so that they can join and perform work on behalf of a collaborative context.

One of the most basic functionality provided by this context is to display the identity of all accessed objects. In that respect, the context must be a **Tracing** context. Also, there are multiple threads that perform work on behalf of the context. Therefore, the context must be **Collaborative** in order support cooperative work and to manage these multiple participants. The **EntrySynchronizing** aspect is necessary for making sure that these participants all start accessing the objects at the same time. **PerformanceMonitoring** is the last aspect that applies to the context. It is not a part of the framework; rather it is specific to this application. It encapsulates the logic for measuring the performance of the system. It intercepts

each call made to the object and, with the help of a timer, it calculates the time elapsed for each method invocation. When the context ends, it displays the results that it has computed (see next part for a sample output).

SAMPLE EXECUTION: The figure 6–1 shows a sample output of the performance measuring application.

```
**** Statistics / Performance Results ****

Total Number of Accesses: 64
Total Elapsed Time: 25065 milliseconds
Efficiency: 391.6 milliseconds per access
-----
The following objects were READ:
accountObject2, accountObject5, accountObject6, accountObject1

Number of Read Accesses: 45
Elapsed Time For Read Accesses: 14298 milliseconds
Read Access Efficiency: 317.7 milliseconds per READ access
-----
The following objects were WRITTEN:
accountObject4, accountObject2, accountObject3

Number of Write Accesses: 13
Elapsed Time For Write Accesses: 8972 milliseconds
Write Access Efficiency: 690.1 milliseconds per WRITE access
-----
The following objects were UPDATED:
accountObject5, accountObject1, accountObject6

Number of Update Accesses: 6
Elapsed Time For Update Accesses: 6194 milliseconds
Update Access Efficiency: 1032.3 milliseconds per UPDATE access

**** Statistics / Performance Results ****
```

Figure 6–1: A Sample Output of the Performance Monitoring Application

The output presents the statistics gathered from the sample application. During a total of 25065 milliseconds, the performance monitoring context has recorded each

and every activity of its participants. These recorded statistics show that read operations are dominant over the write and update operations. This information could be valuable for the application designer; the application could be optimized knowing that majority of accesses are read operations. Furthermore, the output shows the poor performance of update accesses. There were 6 update operations with an average elapsed time of 1032.3 milliseconds. In order to improve the overall performance of the application, the programmer should consider the poor performance of the updates.

INTERACTION OF THE ASPECTS: The sequence diagram (figure 6–2) illustrates how different aspects interact for providing the synchronous entry functionality.

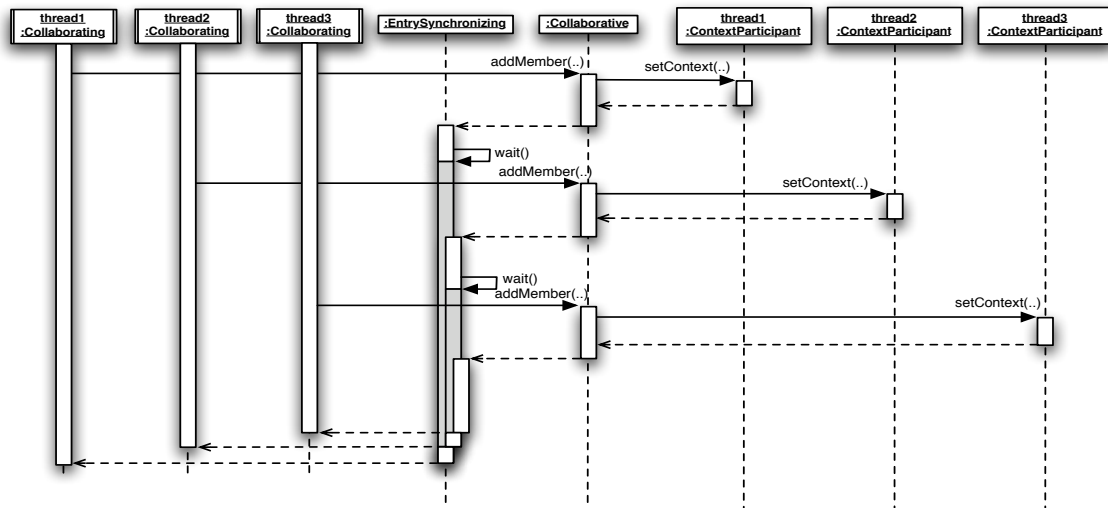


Figure 6–2: Interaction of Aspects for Providing Synchronous Entry

The diagram shows the interaction of aspects as three different threads get ready to perform work on behalf of a context with the synchronized entry requirement. The *Collaborating* aspect initiates the interaction and asks the *Collaborative* aspect

to add thread 1 as one of its participants. *Collaborative* adds the participant to its joined participants list and then notifies the *ContextParticipant* aspect about it. The *ContextParticipant* aspect makes sure that the participant is now linked to the context. When the joining process is done, *EntrySynchronizing* intercepts the call. It checks whether all three threads have joined the context and sees that the answer is no. Therefore, thread 1 gets blocked by the *EntrySynchronizing* aspect.

Next, the *Collaborating* aspect starts the joining process for thread 2. After following similar steps, thread 2 also gets blocked, since *EntrySynchronizing* is still waiting for the last member to arrive. Finally, thread 3 comes in and *Collaborating* & *Collaborative* & *ContextParticipant* carry out the necessary actions for making sure that thread 3 joins the context. After the addition process, *EntrySynchronizing* intercepts the call once again. It checks and sees that all three threads have made it to the entry of the context. Therefore, it stops the blocking operation and releases all the threads at the same time. These three threads are now allowed to perform work on behalf of the context.

6.2 Flat Transaction Model With Optimistic Concurrency Control & Deferred Update

GOAL: The goal of the example is to examine the possibility of creating a context that will match up to the requirements of the flat transaction model. Furthermore, the context must be using optimistic concurrency control for the concurrency management and deferred update as its update strategy.

DESIGN: The table 6-2 summarizes the aspects that are used to construct the model.

Table 6–2: The Design of Flat Transaction Model with Optimistic Concurrency

Aspects		
<i>Object</i>	<i>Context</i>	<i>Thread</i>
Traceable Versioned ContextAware ContextTracing	Tracing Deferring OutcomeAware Recovering OptimisticValidation	OutcomeAffecting

In order to participate and be traced by a transactional context, the object has to be both **ContextAware** and **Traceable**. Furthermore, for providing support for optimistic concurrency control & deferred update by being able to encapsulate several versions of its state, the **Versioned** aspect also has to be applied to the object. Finally, in order to help resolve the conflicts in the validation step the object must trace the contexts that it has interacted with. As a result, the object needs the functionality provided by the **ContextTracing** aspect.

When the participant finishes its work in the context, it must indicate whether the transaction should commit or abort; meaning that it should be **OutcomeAffecting**.

For keeping a list of objects that have been accessed by the transaction, the context has to be a **Tracing** context. **Deferring** must also be applied to the context because it ensures that the context works in complete isolation; a requirement of the optimistic concurrency control. The binary outcome of a transaction (i.e. commit vs. abort) is provided by the **OutcomeAware** aspect. In case of an abort, the context needs the functionality of the **Recovering** aspect: being able to rollback the state changes made on behalf of the transactional context. **OptimisticValidation** is

not an aspect of the framework, instead it should be provided by the application programmer to perform the backward validation step of the transaction. However, all the necessary pieces to carry out the validation step are provided by the aspects of the framework. The read and write lists provided by the two tracing aspects, **ContextTracing** of object and **Tracing** of context, can be used in determining whether a transaction conflicts with other transactions.

SAMPLE EXECUTION: The figure 6–3 displays a sample execution trace of the application.

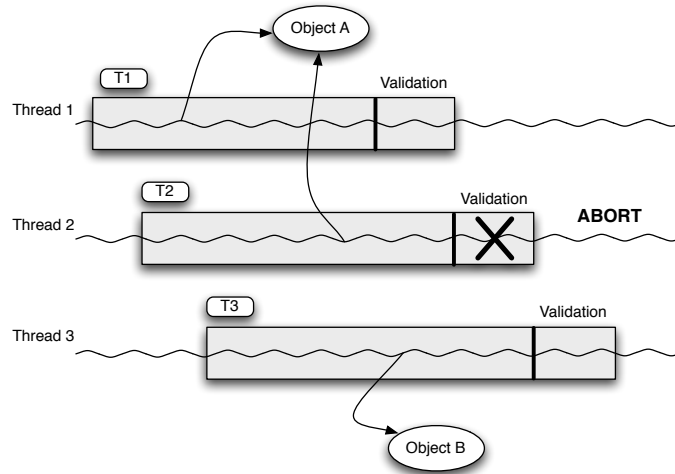


Figure 6–3: Sample Execution Trace for Flat Transaction Model

The trace consists of three transactions that execute concurrently and access two different transactional objects. Transaction 1 reads object A, Transaction 2 performs a write operation also on object A and Transaction 3 executes a write operation on object B (the details of the method invocation process are illustrated in the next section). Thread 1 is the first OutcomeAffecting that votes commit and

therefore Transaction 1 becomes the first transaction to go under validation. Since there aren't any concurrent transactions that have already committed, it gets easily validated and commits. After this, Transaction 2 initiates the validation phase. The *OptimisticValidation* aspect checks the write set of T2 (using the *Tracing* aspect) and notices that object A has been read by T1 (using the *ContextTracing* aspect), which is a conflicting transaction for T2. Therefore, T2 cannot be validated and it is aborted. At this point, the *Recovering* aspect comes into action discarding the local copy of object A that had been created for T2. Later, T3 also enters the validation process and it easily validated since object B is not in any list of the other concurrent transactions. As a result, the *Deferring* aspect makes sure that the T3's version of object B now is to be the main version of the object, hence globally visible.

INTERACTION OF THE ASPECTS: The figure 6-4 presents how a call to a transactional object is intercepted, and how the individual aspects collaborate to provide the desired functionality.

When an operation of a transactional object is invoked, *ContextAware* intercepts the call and identifies the origin of the call. If the call comes from a participant, *ContextAware* contacts the *ContextParticipant* aspect for finding out the context of the participant. When the context information is acquired, *ContextAware* notifies the context about the work that is going to be performed by its participant by calling the method *work*. Right away, the *ContextTracing* aspect intercepts this notification call. It stores the context identity and the access type of the method that the context is about to invoke. The access type - read, write or update - is determined based on the information provided by *AccessClassified*.

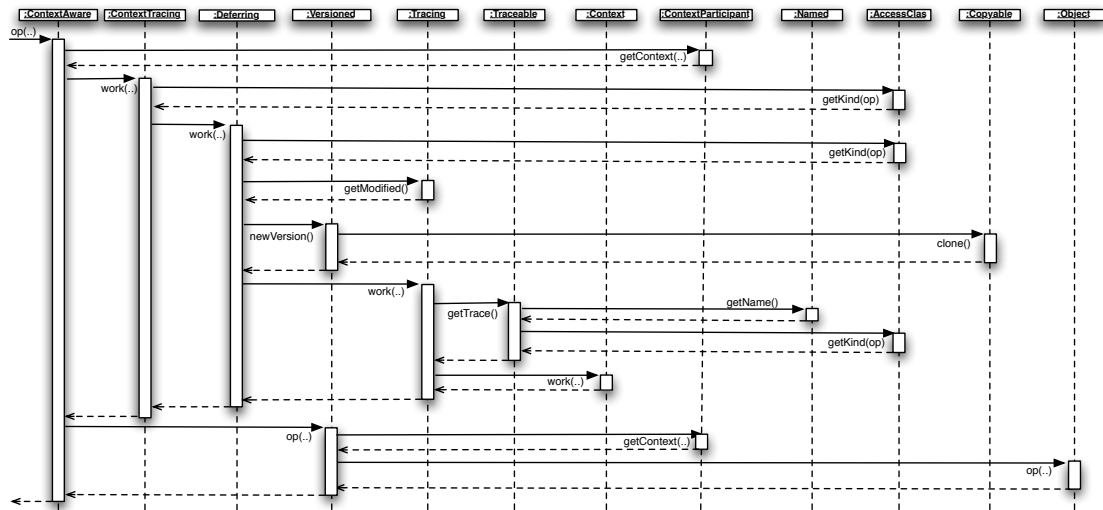


Figure 6–4: Interaction of Aspects for Deferred Update Method Invocation

Next, the *Deferring* aspect intercepts the call. It first checks whether the invoked method will modify the state of an object. Once again, *AccessClassified* provides this information. Seeing that it is a modifying operation, the *Deferring* aspect then tries to find out whether the object was previously modified by the context. This step is carried out by getting a list of modified objects from the *Tracing* aspect. Since the object is not in that list, *Deferring* concludes that it is the first time that the object is modified by the context. As a result, it asks the *Versioned* aspect to create a new version and registers the context for this version of the object. In order to create a new version, *Versioned* asks the *Copyable* aspect to clone the state of the object.

After the *Deferring* aspect, *Tracing* intercepts the notification call. In order to record the access, *Tracing* asks the *Traceable* aspect to create a trace of the object. The trace needs the identity of the object and the access type of the invoked method; these two functionalities are provided by the *Named* and *AccessClassified*

aspects respectively. Finally, the notification call reaches to the context. In this case, the context doesn't need to take further actions, therefore the call simply returns.

Now finally the original call to the transactional object can proceed. The call, however, is again intercepted by the *Versioned* aspect. *Versioned* checks to see whether the call is coming from a certain context: it asks the *ContextParticipant* aspect to provide its context information. Then, *Versioned* determines whether the calling context has a specific view of the object. Since the *Deferring* aspect had previously created a new version and registered the context for that version, *Versioned* performs the method invocation on that corresponding version. After the invocation, the call returns.

6.3 Nested Transaction Model with Pessimistic Concurrency Control & InPlace Update

GOAL: Nested transactions provide a more fine-grained mechanism for handling rollback operations. They allow a certain fraction of a transaction (i.e. child transaction) to be aborted, without enforcing the abortion of the whole transaction (i.e. parent transaction). The goal of the example is to build a context that will represent the nested transaction model. The context should be using pessimistic concurrency control, in this case 2-phase locking, and it should adopt the in-place update strategy.

DESIGN:

Traceable and **ContextAware** apply to the object, providing the basic functionality of being traced by the context. However, since the selected concurrency & recovery scheme is different, the other aspects that are applied to the object are different than in the previous example. Pessimistic concurrency control uses locks to

Table 6–3: The Design of Nested Transaction Model with Pessimistic Concurrency

Aspects		
<i>Object</i>	<i>Context</i>	<i>Thread</i>
Lockable Traceable Checkpointable ContextAware	Tracing Two-Phase-Locking Checkpointing OutcomeAware Recovering Nested	OutcomeAffecting

provide the isolation among the transactions. Therefore, the object should be **Lockable** so that the transactions can lock the object before accessing it. Furthermore, since in-place update works on the main copy of the object, there is a need to save the state of an object for recovery purposes. For that reason, the object should be **Checkpointable** so that the transactions can take a snapshot before performing any modification.

The **OutcomeAffecting** aspect has to apply to the thread entity, so that a thread can affect the outcome of a transaction by submitting its decision.

Storing a list of the objects that have been accessed by a transaction is a fundamental functionality and it is provided by the **Tracing** aspect. In order to ensure the isolation property through the use of locks, the context needs the **Two-Phase-Locking** aspect. Before modifying any object, the context must make sure that it stores the state of the object. In that respect, the **Checkpointing** aspect should be applied to the context. The context relies on the **OutcomeAware** aspect to provide the outcome (i.e. commit vs. abort) of the transaction. With this outcome in hand, the **Recovering** aspect is necessary to guarantee the all-or-nothing property of a transaction: all the state changes performed by the transaction might have to

be rolled back by the Recovering aspect. Finally, in order to nest different levels of work in each other, the **Nested** aspect is required.

It should be noted that in terms of aspects, there is no assistance required from the developer to create nested transactions. With an accurate composition, the aspects of the framework are powerful enough to capture the needs of the nested transactional model with pessimistic concurrency control and in-place update.

SAMPLE EXECUTION: Figure 6–5 displays a sample execution trace of a trip reservation application that uses the nested transaction context.

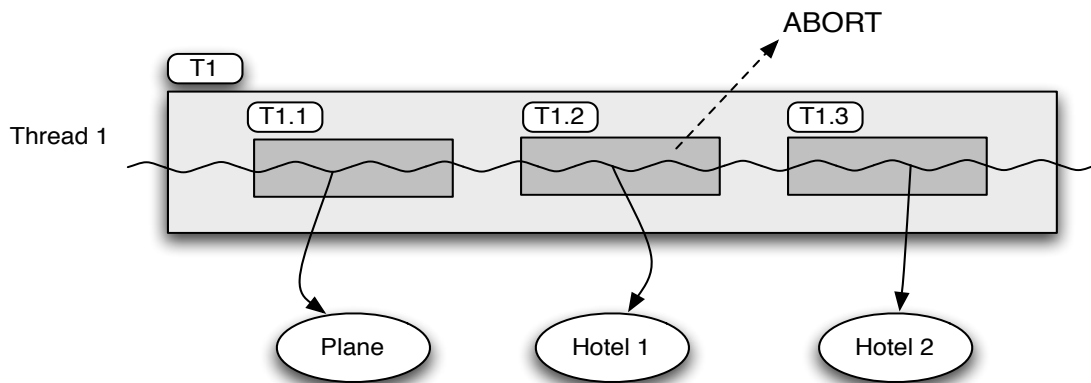


Figure 6–5: Sample Execution Trace for Nested Transaction Model

The execution trace shows the execution of a transaction that ultimately aims to reserve a plane and a hotel for a trip reservation. Thread 1 creates the parent transaction T1. In order to reserve a plane, T1 creates a child transaction T1.1, a functionality provided by the Nested aspect. T1.1 acquires the lock of the plane object and performs the reservation without any problems. When T1.1 commits, the work performed (i.e. acquired locks, traces) are transferred to the parent transaction

T1. Later, T1 creates another child transaction T1.2 for performing the hotel reservation. However, T1.2 cannot perform the operation because it sees that the hotel is already full. Therefore, T1.2 is aborted and the lock that it has acquired is thrown away (the details of transaction abort are explained in the next section). The parent transaction T1 though, continues executing and creates a new sub-transaction T1.3 for reserving another hotel. T1.3 succeeds in reserving the hotel and is committed. Once again, the acquired lock and the generated trace get transferred to the parent transaction T1. Finally, when T1 commits, it releases the locks of the plane and the second hotel object, therefore allowing the effects of the transaction to be globally visible.

INTERACTION OF THE ASPECTS: The sequence diagram shown in figure 6–6 illustrates how aspects work together during the abortion process of a nested transaction.

Not being able to achieve the intended goal, the participant votes abort for the outcome of the transaction. The *OutcomeAffecting* aspect communicates with the *OutcomeAware* aspect to signal that the participant wants to abort the transaction. After the thread votes, it completes its work on behalf of the transaction. As a result, *ContextParticipant* tells Context that the participant should no longer be in the context. Since a context is meaningless without its participants, the removal of the only participant also implies the termination of a context. However, before the context is terminated, several actions must be taken in order to guarantee the correct behavior of a transaction.

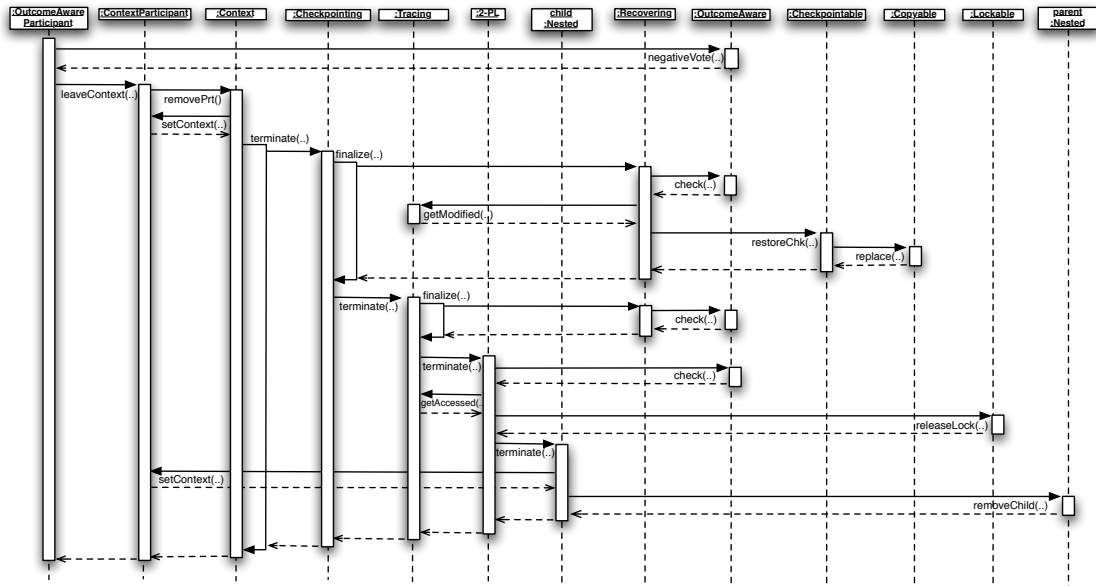


Figure 6–6: Interaction of Aspects for Aborting a Nested Transaction

First, the *Checkpointing* aspect intercepts the call. Assuming that the transaction was a success, it makes sure that the established checkpoint does not get discarded so that the parent transaction can make use of it, if needed. However, seeing that the outcome of the context is negative, the *Recovering* aspect intervenes. It first gets a list of modified objects from the *Tracing* aspect. In order to rollback the state changes in these modified objects, *Recovering* asks the *Checkpointable* aspect for each object to restore the established checkpoints. Checkpointable requires assistance from the *Copyable* aspect for restoring the checkpoints: the current state of the object is replaced by the state of the checkpoint.

Next, the *Tracing* aspect intercepts the termination call. When a child transaction ends, all the work it has performed should be delegated to its parent. In that respect, the *Tracing* aspect ensures that the traces of the child context are transferred

to the parent. However, since the transaction is aborted, this is no longer true. In order to ensure the correct functionality, *Recovering* intervenes once again and checks the outcome of the context. Seeing that it is to be aborted, it blocks the *Tracing* aspect so that the traces don't get transferred to the parent context.

The next aspect that intercepts the call is *Two-Phase Locking*. For ensuring the correct behavior in the presence of the *Recovering* aspect, it first checks the outcome of the context. Noticing the abort decision, it asks *Tracing* to provide the list of accessed objects. In order to rollback the state, it releases all the locks that the child transaction has acquired. Furthermore, it gives back all the locks that it has borrowed from the parent context (no locks were borrowed in this example).

Nested is the last aspect to intercept the termination call. It is responsible for moving the participant back to the parent context. Therefore, it communicates with the *ContextParticipant* aspect for making sure that the participant ends up as a member of the parent transaction. Furthermore, whenever a child context terminates the link to a parent, the parent context must as well be updated. The *Nested* aspect performs this update by removing the terminated context from the parent's list, which keeps references to its children.

6.4 Open Multithreaded Transaction Model with Pessimistic Concurrency Control & InPlace Update

GOAL: Open Multithreaded Transactions are a quite sophisticated transactional model that allows several threads to enter the same transaction in order to perform a joint activity. In that respect, the model controls and structures not only accesses to the objects, but also threads taking part in the transaction. The goal of this example

is to study whether the aspects of the framework could be combined to achieve such a complicated transactional model.

Table 6–4: The Design of OMTT Model with Pessimistic Concurrency

Aspects		
<i>Object</i>	<i>Context</i>	<i>Thread</i>
Lockable Traceable Checkpointable ContextAware Shared	Tracing Two-Phase-Locking Checkpointing OutcomeAware Recovering Nested Collaborative ExitSynchronizing OutcomeVoted	OutcomeAffecting Collaborating

DESIGN: Since the concurrency and recovery schemes are the same as in the previous example, there aren't many changes in the aspects that apply to the objects. The only difference is the addition of the **Shared** aspect. In the OMTT model, the participants of a transaction are allowed to perform cooperative work by sharing the same objects. Therefore, there is a need for the *Shared* aspect to preserve the data consistency of these objects in the presence of concurrent accesses made by the participants.

The threads of an Open Multithreaded Transaction have to be aware of the cooperative work that may be performed in the transaction. The **Collaborating** aspect provides that functionality. Furthermore, the threads are **OutcomeAffecting** so that they can vote on the outcome of the transaction.

The context aspects that are used to configure the concurrency and recovery schemes are also the same as in the previous example. However, more aspects are

added in order to capture the cooperative behavior of the Open Multithreaded Transaction model. Primarily, the **Collaborative** aspect must be applied so that multiple participants may perform work on behalf of the same context. In order to prevent information smuggling, the participants of an Open Multithreaded Transaction are not allowed to leave the transaction until the outcome has been determined. The context needs the **ExitSynchronizing** aspect for providing that functionality. An Open Multithreaded Transaction commits if and only if all of the participants vote commit. The **OutcomeVoted** aspect should be present for keeping track of the votes submitted by the participants, and for modifying the outcome of the context accordingly.

SAMPLE EXECUTION:

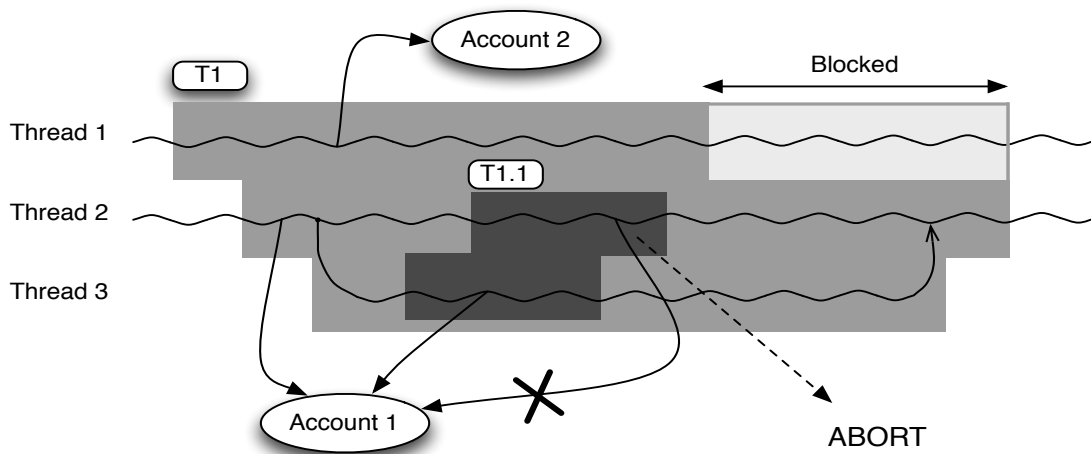


Figure 6-7: Sample Execution Trace for OMTT Model

The figure 6-7 shows a sample execution trace of an Open Multithreaded Transaction model. The transaction T1 is created by thread 1 and later thread 2 joins

this transaction. Thread 2 starts performing work by accessing the Account1 object (the details of this invocation are provided in the next subsection). Right after that, thread 2 creates a new thread named thread 3. Collaborating makes sure that the newly created thread becomes a spawned participant of the Collaborative context. Concurrently, T1 performs more work: thread 1 executes an operation on the Account2 object.

Next, thread 3 creates a new transaction T1.1 and the *Nested* aspect makes sure that the new transaction becomes a child of T1. The spawned participant of T1, thread 3, decides to become a joined participant of T1.1. It also performs some work by accessing the Account1 object. At the same time, thread 2 also joins this nested transaction T1.1. However, even before thread 2 gets to perform work, thread 3 votes abort for the outcome of the transaction. In the Open Multithreaded Transaction Model, each participant has to vote commit in order for the transaction to commit. If only one participant votes abort, the transaction is aborted. Therefore, *OutcomeVoted* makes sure that the abort decision is made and terminates the transaction using the *Terminatable* aspect. Since the outcome of the transaction is determined, thread 3 simply leaves the transaction and ends up in T1 in accordance with the nesting rules. After this point, no further work can be performed on behalf of transaction T1.1. Therefore, when thread 2 tries to access Account1 object, the *Terminatable* aspect blocks the call. Thread 2 is notified of the outcome, is removed from T1.1 and ends up in T1.

Meanwhile, thread 1 votes commit for the outcome of transaction T1. Since the outcome of T1 is not determined yet, the *ExitSynchronizing* aspect makes sure that

thread 1 is blocked. Later the spawned participant also votes commit and terminates immediately. Finally, thread 2 also votes commit and the outcome of the transaction is determined as commit.

INTERACTION OF THE ASPECTS: The sequence diagram in figure 6–8 captures the interaction of the aspects when a method of the Account1 object is invoked by Thread2.

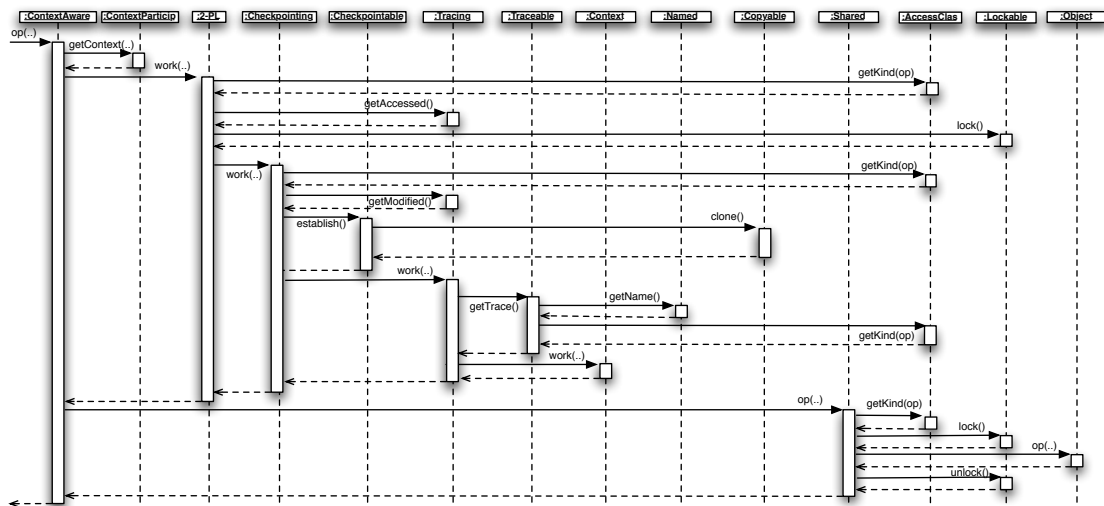


Figure 6–8: Interaction of Aspects for InPlace Update Method Invocation

Whenever a method of a transactional object is invoked, *ContextAware* intercepts the call and identifies the origin of the call. If the call comes from a participant, *ContextAware* contacts the *ContextParticipant* aspect to determine the context of the participant. Finally, *ContextAware* notifies the context of the participant about the work that it wants to perform.

Two-Phase Locking is the first aspect to intercept the notification call. In order to find out whether the context has accessed this object before, it asks the *Tracing*

aspect to provide a list of all accessed objects. By checking this list, it sees that it is the first time that the context executes an operation on the object. Therefore, the *Two-Phase Locking* aspect tries to obtain the object's transactional lock, which is provided by the *Lockable* aspect. The kind of lock (read, write or update) is chosen based on the information provided by *AccessClassified*. Before granting the lock, the *Lockable* aspect verifies that this lock does not conflict with other transactions in progress. If a conflict is detected, the participant requesting the lock is blocked and has to wait for the release of the conflicting lock. Otherwise, the lock is granted and the call is allowed to proceed.

Next, the in-place recovery scheme depends on the *Checkpointing* aspect to intercept the call and to checkpoint the state of the transactional object before it is modified. To find out whether the invoked operation is actually a state modifier method, *Checkpointing* consults the *AccessClassified* aspect. Learning that the method is indeed going to modify the state of the object, *Checkpointing* tries to find out whether the context had previously established a checkpoint for this object. To do that, *Checkpointing* needs the list of objects that the context had previously invoked modified operations on. *Checkpointing* obtains this list from the *Tracing* aspect. Since it is the first time that this object is modified, *Checkpointing* asks the *Checkpointable* aspect to checkpoint the state of the object. For establishing a checkpoint, the *Copyable* aspect clones the state of the object and the returned state is stored as a checkpoint by the *Checkpointable* aspect.

Tracing is the next aspect to intercept the call and it records the access made by the context. It asks the *Traceable* aspect to create a trace of itself. The trace

consists of two pieces of information: the identity of the object and the access type of the invoked method. *Traceable* gathers this information from the *Named* and *AccessClassified* aspects and prepares the trace to be stored by the *Tracing* aspect. Finally, the notification call reaches the context. However, the context doesn't perform any specific action and simply returns the call.

As the context notification call comes to an end, the original call to the transactional object is intercepted by the *Shared* aspect. The aspect makes sure that no two threads are modifying the object's state concurrently. Learning the type of access from the *AccessClassified* aspect, *Shared* acquires the appropriate shared lock from the *Lockable* aspect. After the method has been executed, *Shared* releases the mutual exclusion lock.

CHAPTER 7

Aspect Interferences in AspectOPTIMA

7.1 Definition of Aspect Interference

Aspect-Oriented programming has been successful in improving the modularization of software in the presence of crosscutting concerns. A key point when dealing with aspect-oriented programming is the notion of aspect interference. The so called aspect interference problem is considered to be one of the remaining challenges of aspect-oriented software development: the composition of the independently programmed aspects may cause emerging conflicts due to unexpected semantic interactions.

Since multiple aspects are implemented separately, there could be problems when the behavior of these aspects are added to the base program. When multiple aspects apply to the same join point, different composition orders may give rise to various inconsistency problems. In such circumstances, the aspects interfere with each other in a potentially undesired manner; there could be side-effects caused by the other aspects as several aspects work with the state of the base program simultaneously. Therefore, when multiple aspects are composed, one has to make sure that the outcome is semantically correct and the intended behavior is preserved.

[DBs06] presents a nice example of the aspect interference concept. In the example, there is a base application that contains a simple protocol for sending and receiving data. Two aspects, encryption and logging apply to the this base

application. The encryption aspect is responsible for encrypting all outbound traffic and decrypting all the inbound traffic, and the logging aspect takes care of logging sent and received data. Both of these aspects apply to the same join point in the base application (sending and receiving of data), and the ordering of these aspects creates the interference. If the logging aspect is used for debugging purposes, it should only be applied to non-encrypted data. Therefore, the logging aspect should precede the encryption aspect, so that the data is logged before it is encrypted. On the other hand, in a hostile environment, it would be dangerous to log sensible data in clear text. In such a domain, the encryption aspect should be activated earlier, so that the data is encrypted before it is logged.

The example, explained in the previous paragraph, presents a case where both orderings of the aspects make sense. The programmer may select the ordering that would fit the needs of its application. However, in some other interfering aspects, it could well be the case that only one ordering is acceptable. In other words, there could only be a single ordering that would resolve the interference between the aspects.

7.2 Resolving Interferences in a Reusable Way

The thesis focuses on the design of AspectOPTIMA, which is a pure aspect-oriented framework. It contains 28 aspects that may be applied to the base application. However, some aspects conflict with each other and therefore the composition of these aspects turns out to be a delicate process. Based on the experience gained by designing such a framework, the thesis proposes a novel technique that resolves these aspect interferences in a reusable way.

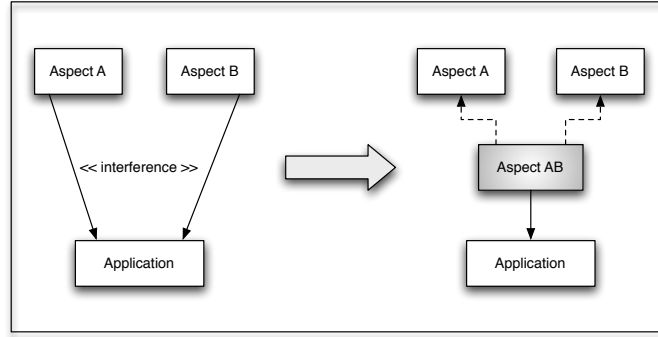


Figure 7–1: A Technique to Resolve Aspect Interferences

Figure 7–1 shows the technique that has been used to resolve interferences in the framework. The approach is as follows: For each conflicting pair of aspects, one needs to create a new **interference aspect** that takes care of the conflict. Let's assume that the composition of **aspect A** and **aspect B** results in an interference. In that case, the **aspect AB** is used to ensure the correct behavior of the application. The **aspect AB** depends both on **aspect A** and **aspect B**, and also encapsulates the interference logic between these two aspects. Therefore, whenever a conflict occurs, **aspect AB** is capable of resolving the conflict by properly configuring **aspect A** and **aspect B**. The use of **aspect AB** is not limited to a specific application. It has to be re-used in any context that contains **aspect A** and **aspect B**, to solve the interference of these aspects.

7.3 Interfering Aspects in AspectOPTIMA

This section is dedicated to the analysis of the interfering aspects in the AspectOPTIMA framework. In order to resolve these interferences, the thesis makes use of the novel technique that was explained in the previous section. Basically, the

approach is to create a new interference aspect for each conflicting pair of aspects in the original design. The following part presents these newly created aspects along with a description of how it handles the interferences:

CopyableLockable - The use of locks is a mechanism to control the accesses made to the state of an object. When the Copyable aspect clones the object, the locks of an object (introduced by the Lockable aspect) should *not* be duplicated. Instead, the users of this object should acquire new locks for accessing this newly duplicated state.

CopyableShared - In order to preserve data integrity, the state of a Shared object should only be copied when no other thread is making modifications.

SerializableNamed - In the presence of the Named aspect, the object's name should also be serialized with its state.

VersionedContextAware - Versioned aspect only allows individual threads to create and work on specific versions of the object. In the presence of the ContextAware aspect, it should be possible for a context to do the same; register itself to a version of the object and keep working on that version.

PersistableCheckpointable - When a Checkpointable object is made persistent, all checkpoints have to be made persistent as well.

PersistableVersioned - Persistent should know how to handle the presence of Versioned, since in general, there is a "main" version that contains the state of the object that is currently considered the correct one.

TracingNested - In the presence of the Nested aspect, Tracing should be aware of the child-parent relationship among the contexts. Before a child context terminates, it should share its traces with its parent.

DeferringNested - Deferring contexts have a separate view of the transactional objects that they access. When Deferring and Nested aspects are used together in an application, there is a need to transfer the view from a child context to its parent context. This is to ensure that the work performed by a child gets delegated to its parent upon termination of the child context. Furthermore, during its lifetime, a child context should be able to access the view of its parent.

CheckpointingNested - When a context terminates, the Checkpointing aspect assumes that the performed work was a success. Therefore, it discards all the checkpoints that the context has established during its lifetime. However, in the presence of nested contexts, there may be a need to keep these checkpoints. If the parent context hasn't already established a checkpoint for a object, then the child context should not discard the object's checkpoint so that the parent may make use of it.

TwoPhaseLockingNested - When the Two-Phase Locking aspect is used in a Nested environment, there is a need to handle the borrowing and lending of locks. When necessary, a child context is allowed to borrow locks from its parent. Furthermore, when the child context terminates, all of its locks (borrowed or newly acquired) should be returned to the parent context.

TwoPhaseLockingNestedRecovering - In the presence of the Nested aspect, the default behavior of Two-Phase-Locking is making sure that a child context

transfers all of its locks to its parent upon termination. However, if Recovering is involved, this behavior would no longer be correct. As a part of the recovery process, the `TwoPhaseLockingNestedRecovering` aspect makes the distinction between the new locks that the child context has acquired and the locks that it has borrowed from its parents. In case of an abort, `TwoPhaseLockingNestedRecovering` ensures that these new locks get released instead of being transferred to the parent.

PausableExitSynchronizing - In order to provide the correct functionality, the `Pausable` aspect must be aware of the participants that are waiting at the exit of a context. Whenever a pause request is issued, `Pausable` is responsible for blocking the participants of the context. However, some participants could have already been blocked by the `ExitSynchronizing` aspect. Therefore, `Pausable` must communicate with `ExitSynchronizing` to determine the number of the participants that are blocked at the exit.

PausableEntrySynchronizing - Similarly, the `Pausable` aspect must also be aware of the participants that are waiting at the entry of a context. In that respect, `Pausable` should contact `EntrySynchronizing` aspect to retrieve the number of blocked participants.

CHAPTER 8

AspectOPTIMA Implementation Comments

8.1 Implementation Platform

AspectOPTIMA framework was implemented in AspectJ [KHH⁺01], which is an aspect-oriented extension of the Java programming language. AspectJ emerged from a research work at Xerox PARC that aims at modularizing crosscutting concerns and is currently considered to be the most mature AOP implementation. This section briefly introduces the fundamental concepts and constructs of AspectJ that were used in the implementation.

AspectJ supports two types of crosscutting behaviors: dynamic and static crosscutting. These crosscutting behaviors 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 declarations, but it cannot be explicitly instantiated.

Dynamic Crosscutting: Dynamic crosscutting techniques are used for defining behaviors that modify the runtime execution of a system either by augmenting or replacing it. There are 3 constructs that may be used to achieve the dynamic behavior.

- **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.

- **POINTCUT** - A pointcut is a construct used for capturing join points of interest and their associated context - such as the current executing object, the target object of a call or execution and the arguments of the join point. 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.
- **ADVICE** - An advice defines the actions to be taken at the join point(s) captured by a pointcut. AspectJ supports three types of advice: the before, the after and the 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 joinpoint and has the ability to augment, bypass or allow its execution.

Static Crosscutting: The static crosscutting constructs are used for modifying the static structure (i.e., classes, interfaces and aspects) of a system.

- **INTERTYPE DECLARATIONS** - The member introduction concept of AspectJ is a feature that has been extensively used in the implementation of the AspectOPTIMA framework. This concept facilitates the introduction of data members and methods - with implementation - into classes and interfaces.

A publicized use of introduction is in the provision of a default interface implementation that is automatically “added” to each implementing class.

- **MODIFICATION OF 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.

There were some limitations/difficulties that were encountered during the implementation of the framework. However, the evaluation of these limitations is beyond the scope of the thesis. [DE06] provides a detailed discussion that analyzes the implementation of the old framework using AspectJ.

8.2 Achieving Reusability

According to the design of the framework, the functionality of some aspects can only be achieved through the assistance provided by other aspects (e.g., *Checkpointable* depends on *Copyable* aspect to duplicate its state). As a result, aspects require a way for expressing their need of the functionalities provided by other aspects (inter-aspect configurability). In addition, the aspects were carefully constructed so that they could be individually reused by any class in any context. Therefore, a way has to be found to make it possible for a developer to specify, outside of the framework, to which objects or classes an aspect should be applied to (separate aspect binding).

The abstract introduction idiom [HU03] has been proposed as a strategy for implementing aspects that can be reused in different contexts. It allows the encapsulation of a concern into one unit, however binding of this concern to a target class is deferred until weave-time. This strategy has three participants (Figure 8–1)

- Introduction Container: It is a construct that is used as the target for the inter-type member declarations.
- Introduction Loader: It is the aspect that introduces crosscutting behaviors to the introduction container.
- Container Connector: It is the aspect used for connecting the introduction container to the base application classes.

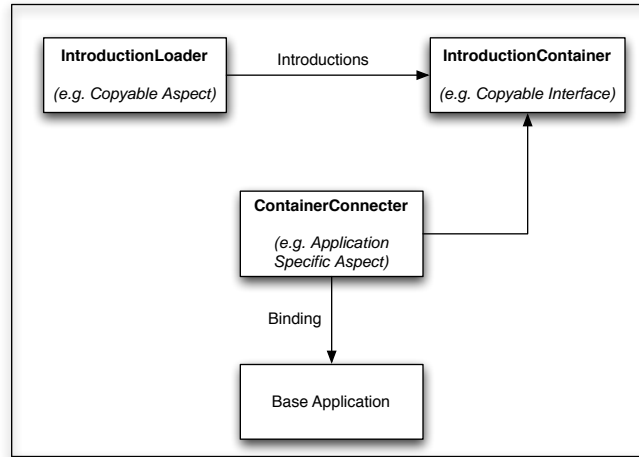


Figure 8–1: Abstract Introduction Concept

The AspectOPTIMA implementation contains a dummy interface (introduction container) for each of the aspects (introduction loader) of the framework. For instance, the interface *Copyable* is associated to the aspect *CopyableAspect*, *Versioned*

is associated to *VersionedAspect* and so on. Each aspect applies its functionality to these dummy interfaces using the intertype declaration technique. As a result, any class that implements one of these interfaces acquires the functionality that is provided by the corresponding aspect. Use of interfaces in achieving the binding of an aspect to a class satisfies the oblivious property [FF00], since developers do not have to modify their systems to accommodate these aspects and the base application is completely ignorant of their existence.

8.3 The Context Object & Context Manager

The design of the AspectOPTIMA framework contains the idea of a context, which represents a well-defined zone of computation. A context is dynamically created by the participants, and it terminates when it no longer contains any participants. Just like any other object, it has a life cycle: it is created, it performs some work and then it is destroyed. Also, multiple contexts can be active at a given time. Therefore it was decided that the context should be represented as an object at the implementation level. Consequently, the `ContextObject` class is responsible for encapsulating the concept of a context.

For creating a customized context, the application programmer must first have a class that extends the `ContextObject` class. Then, the programmer must select the aspects that will be re-composed to construct a customized context. Finally, those selected aspects must be applied to the application class that extends the `ContextObject` class.

In certain applications, there may be a need to create different *type* of contexts. For example, depending on external parameters, the application may select

between a flat transaction model and a nested transaction model. That requires the implementation to support the creation of any type of context. Our implementation supports this by allowing the programmer to create, for each type of context, a new class that extends the `ContextObject` class as described above. At run-time, the `ContextManager` class can, provided the name of the class create a new customized context object for the application.

8.4 Framework API

The current implementation makes sure that the developers do not have to know about the inner workings of the framework. The only dynamic interaction between the programmer and the framework takes place through the application programming interface (API). As a result, the programmer does not have to be bothered with implementation details.

The framework API must be used whenever there is a need to perform any kind of operations on contexts. Since the API is mostly used by the threads, implementation choice was to embed the API inside the aspects of the thread. These aspect inject the methods of the API into the thread entity, allowing the thread to freely use the API methods. For example, the *ContextParticipant* aspect enables a thread to use *createContext()* and *leaveContext()* methods. In collaborative contexts, the *Collaborating* aspect lets a thread use the *createOrJoinContext()* method of the API, so that it could join an ongoing context. Depending on the capabilities of the threads, they can perform more sophisticated operations such as affecting the outcome of a context.

CHAPTER 9

Related Work

9.1 ACTA Framework

ACTA [CR90] is another transaction framework that allows formal reasoning about the properties of transaction models. It is not a transaction model, but rather it can be used to specify new transaction models by determining the effects of transactions on other transactions and the effects of transactions on objects. ACTA uses first-order logic to capture properties of transactions, such as visibility, consistency, recovery, and permanence. Its main building blocks are history, intertransaction dependencies, transaction view, conflict set, and delegation, which can be used to create new transaction models or just extend existing models by methodically modifying their definition.

The transaction *view set* controls the visibility of objects on the transaction, while the *access set* maintains the effects on objects by the transaction. When a transaction is initiated, it is associated with a view set, which virtually contains all the objects accessible to the transaction. Rules for composing the view set are determined by the specific transaction model. When an object in the view set of a transaction is accessed by the transaction, the object becomes a member of the access set. When a transaction commits, every object in its access set is either made persistent or delegated to another transaction. The *delegation* causes a transaction to delegate to another transaction the responsibility of finalizing its effects for some

of the objects in its access set. The delegation enables other transactions to access tentative or partial results of a transaction, and cooperate with one another.

The framework's power lies in the ability to represent structural behavior of transactions and the dependencies between them. However, implementation of ACTA models are not always straightforward. ACTA is a purely meta-model, and therefore does not concern itself with possible implementation issues. Consequently, it is really hard to demonstrate that an ACTA model does in fact capture the semantics of a transactional model. On the other hand, AspectOPTIMA framework serves for a more practical purpose. Using the aspects, which vaguely correspond to the transaction primitives in ACTA framework, it is possible to create various extended models and illustrate the workings of these models.

9.2 Modularization of Advanced Transaction Management

The Ph.D thesis [Fab05] of Johan Fabry aims to provide a successful modularization of advanced transaction mechanisms (ATMS). The contributions made by the thesis can be broken down into three different parts. First, the thesis proposes a new aspect language, KALA, which helps to express a wide variety of ATMS and also allows the modularization of the different concerns contained within such an ATMS. KALA is not model-specific, but a general aspect language that is based on the ACTA [CR90] formal model. Therefore, it allows for a large variety of ATMS to be constructed. Secondly, on top of this aspect language, the thesis builds a family of Domain-Specific languages for ATMS, where each language reifies the concepts exposed by the corresponding ATMS. These domain-specific languages make things more understandable for application programmers since these languages allow

writing code at a high level of abstraction. Third and finally, the thesis shows the extensibility of its solutions by creating a new ATMS and writing KALA code for it, along with a DSL for that language.

Alongside the above research contributions, the thesis also provides two technical contributions. First, based on the ACTA formal model it creates a general Transaction Processing Monitor that supports a wide variety of ATMS. Second, it provides an implementation of an aspect weaver for the KALA language, which generates code using this interface.

As opposed to AspectOPTIMA framework, the level of separation of concerns that the thesis provides remains to be rather limited to the implementation. The thesis contains a central transaction processing monitor (ATPMos) and in order to communicate with this unit, the communication logic (demarcation code) must be inserted into the application. The thesis achieves the separation of concerns by taking out this demarcation code from the core application logic. So, instead of a conceptual separation, the obliviousness provided by the thesis ends up being syntactic.

9.3 Framework of Concurrency Patterns and Mechanisms

The framework of concurrency patterns [CSM06] presents a collection of reusable aspect-oriented implementations of several concurrency control patterns and mechanisms in AspectJ. The authors illustrate how abstract pointcut interfaces and annotations can be used in implementing one-way calls, synchronization barriers, reader/writer locks, scheduler, active objects and futures. The paper also compares the performance overhead, reusability and the unpluggability between conventional

object-oriented implementations and AOP implementations. In the end, it is concluded that the AspectJ implementation is more reusable and pluggable, however it incurs a noticeable performance overhead.

As opposed to AspectOPTIMA, concurrency patterns framework focuses more on the implementation level. It analyzes the capacity of AspectJ in implementing various concurrency patterns and compares such an implementation with the traditional object-oriented approach. On the other hand, AspectOPTIMA framework provides a language-independent design that captures a highly sophisticated decomposition process.

There are also some differences in the implementation of AspectOPTIMA and concurrency patterns framework. In the latter, 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). In the AspectOPTIMA framework implementation however, the only work required by developers to acquire the functionality provided by the aspects is to bind their application classes to the appropriate aspects. 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.

9.4 Persistence Framework

Persistence framework [RC03] represents an effort for supporting persistence by implementing a reusable and oblivious aspect-oriented framework in AspectJ. Using a database application as an example, the authors incrementally demonstrate how reusable aspects can be implemented for database connections, data storage and updates, data retrieval and data deletion. With the experience gained from the database application example, the authors of the framework conclude that persistence aspects could be re-used in other contexts. However, application developers could only be partially oblivious to these aspects since persistence has to be taken into account as an architectural decision during the design phase.

Unlike the Persistable aspect in the AspectOPTIMA framework that relies on other well-defined reusable aspects (Serializeable, Copyable and Named), the persistence provided in [RC03] heavily relies on other database specific aspects that cannot be reused in a non-database persistent context. The implementation of the framework shares some of the drawbacks that were identified during the analysis of the concurrency patterns framework [CSM06]. The developers must also provide concrete pointcuts for each of the abstract pointcuts in the persistence framework to have their applications advised - interfering with obliviousness.

9.5 Framed Aspects

The Framed Aspects [NAWS03] demonstrate how AOP can benefit from parameterization, generation and generalization that frame technology brings. The authors illustrate how frames can enhance reuse and ease the integration and creation of new features and believe that the same technique can be applied to different concerns.

The paper proposes framed aspects as a technique and methodology which combines the respective strengths of AOP, frame technology and Feature-Oriented Domain Analysis (FODA). Using a generic caching component as an example, authors show that framed aspects could be utilized in the creation of reusable component libraries or domains which require high levels of reuse.

Conceptually, there is some overlap between the ideas framed aspects and the design of AspectOPTIMA framework. They both encourage the breaking down of concerns into smaller and smaller units, knowing that aspects could also have different concerns within themselves. AspectOPTIMA applies this idea into the domain of transactions. On the other hand, framed aspects is more interested in the integration of aspects into the applications. The authors claim that framed aspects could be used to allow configuration of reusable aspects that can be woven into existing systems where the original code may or may not be available, thus allowing frame techniques to be used in legacy systems to some degree.

CHAPTER 10

Conclusion

10.1 Summary of Results

The research carried out in [KG06] presents a challenge case study for the aspect-oriented community. It decomposes the ACID properties (atomicity, consistency, isolation, durability) of transactions into a set of ten base aspects, each one providing a well-defined reusable functionality. It then shows how these ten aspects can be configured and composed in different ways to achieve various concurrency controls and recovery schemes. Spotting the innovative approach of such a research, the goal of the thesis was to take the case study one step further.

The thesis took the necessary steps to incorporate concerns of transactional life cycle management into the framework; a piece that was missing in the original case study. Since the object and thread entities can not encapsulate these new concerns, the first step was to propose a new *context* entity to act as the bridge between the objects and the threads. The context is basically a region of computation with a well-defined state, making it a perfect candidate for encapsulating the transaction management concerns.

With the introduction of the context entity, the existing framework had to go through a change. The thesis refactored the framework so that the object and the threads were now aware of this new context idea. Furthermore, the thesis enhanced the design of the framework by adding new aspects that serve as the building blocks

for transactional models. The new AspectOPTIMA framework now enables the design of various transactional models through the configuration and composition of these new aspects.

The decomposition process, which was performed for the new design, sticks to the properties of the original case study:

- **Clear separation of concerns:** Each aspect provides a well-defined functionality. For example, *Shared* takes care of ensuring mutual exclusion of state updates.
- **High reusability:** Each aspect can be used in other applications in a stand-alone way to implement similar functionality. For example, *Recovering* can be used in any application that wants to automatically recover state changes made by an unsuccessful unit of work.
- **Complex aspect dependencies:** Some aspects cannot function properly without the functionality offered by other aspects. For example, *Checkpointing* depends on the presence of *Tracing*. *Checkpointing* consults the *Tracing*'s list of modified objects for deciding whether a new checkpoint should be established or not.
- **Complex aspect interference:** Some aspects have to adapt their functionality in the presence of other aspects. For example, *Tracing* has to detect the presence of *Nested*, and make sure that upon termination the child context shares its list of accessed objects with its parent context.

As the last item points out, the framework includes aspects that have complex interferences among each other. These interferences have been resolved, by a novel

reusable technique that has been proposed by the thesis. Finally, in order to verify the correctness of the AspectOPTIMA framework, the new design has been implemented using AspectJ [KHH⁺01] as the programming language.

10.2 Future Work

As any research study, this work conceptually opens avenues for future work. This section discusses these avenues by presenting four major extension points for the thesis.

- Concurrency control and recovery can be further enhanced when more information about the semantics of operations is available. The AspectOPTIMA framework could be expanded to include a *SemanticClassified* aspect that defines forward and backward commutativity for all operations of an object, and maps every operation to a corresponding inverse operation. Such semantic information opens the door to semantic-based concurrency control [RC96] and logical recovery based on intention lists.
- Certain fault tolerance models, such as Open Multithreaded Transactions [Kie03a] or Atomic Actions [LA90], incorporate disciplined exception handling mechanisms. One future work would be analyzing the exception handling strategies in these models and decomposing them into reusable aspects. The addition of these new aspects into the AspectOPTIMA framework would allow the construction of contexts that are now capable of controlling the raised exceptions.
- A future work that may bring a new dimension to the framework is the introduction of dependencies among the contexts. Currently, except the nested hierarchy, there is no relationship between different contexts. However, in order

to encapsulate the needs of certain extended transactional models such as Sagas [GMS87], there is a need to handle various kinds of dependencies among the contexts. Managing these dependencies would also be a giant step in designing contexts that adopt Look-Ahead [MKR06] optimization principles.

- The ultimate goal of the AspectOPTIMA project is to create a framework that allows the creation of any fault tolerance model. Apart from supporting transactional models, the framework could also handle models such as recovery blocks and n-version programming. In order to achieve this ambitious goal, the fault tolerance domain must be well analyzed to determine what additional aspects have to be incorporated into the framework.

REFERENCES

- [Boo94] Grady Booch. *Object-oriented analysis and design with applications (2nd ed.)*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.
- [CR86] Roy H. Campbell and Brian Randell. Error recovery in asynchronous systems. *IEEE Trans. Softw. Eng.*, 12(8):811–826, 1986.
- [CR90] Panos K. Chrysanthis and Krithi Ramamritham. Acta: A framework for specifying and reasoning about transaction structure and behavior. In *SIGMOD Conference*, pages 194–203, 1990.
- [CSM06] Carlos A. Cunha, João Luís Sobral, and Miguel P. Monteiro. Reusable aspect-oriented implementations of concurrency patterns and mechanisms. In *AOSD*, pages 134–145, 2006.
- [DBs06] P. E. A. Durr, L. M. J. Bergmans, and M. Akşit. Reasoning about semantic conflicts between aspects. In R. Chitchyan, J. Fabry, L. M. J. Bergmans, A. Nedos, and A. Rensink, editors, *Proceedings of the First Aspect, Dependencies, and Interactions Workshop*, pages 10–18. Lancaster University, July 2006.
- [DE06] Ekwa J. Duala-Ekoko. Evaluating the expressivity of aspectj in implementing a reusable framework for the acid properties of transactional objects. Master’s thesis, School of Computer Science, McGill University, Montreal, Canada, 2006.
- [EGLT76] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.
- [Fab05] Johan Fabry. *Modularizing Advanced Transaction Management — Tackling Tangled Aspect Code*. PhD thesis, Programming Technology Lab (PROG), Vrije Universiteit Brussel, Brussels, Belgium, 2005.

- [FF00] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness, 2000.
- [FS96] Paulo Ferreira and Marc Shapiro. Larchant: Persistence by reachability in distributed shared memory through garbage collection. In *Sixteenth International Conference on Distributed Computer Systems*, Hong Kong, 1996.
- [GMS87] Hector Garcia-Molina and Kenneth Salem. Sagas. In *SIGMOD Conference*, pages 249–259, 1987.
- [GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
- [HKM⁺94] Nicholas Haines, Darrell Kindred, J. Gregory Morrisett, Scott M. Nettles, and Jeannette M. Wing. Composing first-class transactions. *ACM Trans. Program. Lang. Syst.*, 16(6):1719–1736, 1994.
- [HU03] Stefan Hanenberg and Rainer Unland. Parametric introductions. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 80–89, New York, NY, USA, 2003. ACM Press.
- [KG02] Jörg Kienzle and Rachid Guerraoui. Aop: Does it make sense? the case of concurrency and failures. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 37–61, London, UK, 2002. Springer-Verlag.
- [KG06] Jörg Kienzle and Samuel Gélineau. Ao challenge - implementing the acid properties for transactional objects. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 202–213, New York, NY, USA, 2006. ACM Press.
- [KHH⁺01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.
- [Kie03a] Jörg Kienzle. *Open Multithreaded Transactions: A Transaction Model for Concurrent Object-Oriented Programming*. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

- [Kie03b] Jörg Kienzle. Software Fault Tolerance: An Overview. In Jean-Pierre Rosen and Alfred Strohmeier, editors, *Ada-Europe*, volume 2655 of *Lecture Notes in Computer Science*, pages 45–67. Springer Verlag, 2003.
- [KLM⁺97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.
- [KR81] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.
- [KRS00] Jörg Kienzle, Alexander Romanovsky, and Alfred Strohmeier. A framework based on design patterns for providing persistence in object-oriented programming languages. Technical Report EPFL-DI No 2000/335, 2000.
- [KS99] Jörg Kienzle and Alfred Strohmeier. Shared recoverable objects. In *Ada-Europe '99: Proceedings of the 1999 Ada-Europe International Conference on Reliable Software Technologies*, pages 397–411, London, UK, 1999. Springer-Verlag.
- [LA90] P. A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1990.
- [Lad03] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.
- [LS79] B. W. Lampson and H. E. Sturgis. Crash recovery in a distributed data storage system. Technical report, 1979.
- [MKR06] Maxime Monod, Jorg Kienzle, and Alexander Romanovsky. Looking ahead in open multithreaded transactions. In *ISORC '06: Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*, pages 53–63, Washington, DC, USA, 2006. IEEE Computer Society.
- [Mos81] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, 1981.

- [NAWS03] L. Neil, R. Awais, Z. Weishan, and J. Stan. Supporting product line evolution with framed aspects, 2003.
- [RC96] Krithi Ramamritham and Panos K. Chrysanthis. *Executive Briefing: Advances in Concurrency Control and Transaction Processing*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.
- [RC03] Awais Rashid and Ruzanna Chitchyan. Persistence as an aspect. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 120–129, New York, NY, USA, 2003. ACM Press.
- [RSB⁺98] Dirk Riehle, Wolf Siberski, Dirk Bäumler, Daniel Megert, and Heinz Züllighoven. Serializer. In Robert Martin, Dirk Riehle, , and Frank Buschmann, editors, *Pattern Languages of Program Design 3*, pages 293–312. Addison-Wesley, 1998.