Requirements Specification of Business Transactions in Use Cases

Kianoush Torkzadeh

A thesis

in

Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of
Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

June 2007

© Kianoush Torkzadeh, 2007

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-34653-2
Our file Notre référence
ISBN: 978-0-494-34653-2

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Abstract

Requirements Specification of Business Transactions in Use Cases

Kianoush Torkzadeh

Functional requirements and the involved user interaction are dependent on the chosen concurrency management strategy of the encompassing business transaction. Use cases are the specification technique of choice for functional requirements documentation. Unfortunately, the current state of the art in use case writing does not provide a proper means to express business transaction specifications. In order to overcome this shortcoming we adapt the use case notation to allow business transactions to be defined in terms of its boundaries, affected resources and the chosen concurrency management strategy. The practicality of our extension is illustrated by a comprehensive case study. Lessons learnt from the case study are presented in the form of a set of heuristics and practical guidelines.

Acknowledgments

I would like to express my deep and sincere gratitude to my supervisor, Dr. Patrice Chalin. His wide knowledge and his logical way of thinking have been of great value to me. His understanding, encouragement and personal guidance have been instrumental in the writing of the thesis.

I warmly thank Daniel Sinnig, for his valuable advice and friendly help.

I owe my loving thanks to my wife Laleh Mousavi Eshkevari. Without her encouragement and understanding, it would have been impossible for me to finish this work.

In addition, I would thank all DSRG members in my lab, for providing valuable comments for my research work.

Table of Contents

Abstract ..... **iii**
Acknowledgments ..... **iv**
Table of Contents ..... **v**
List of Figures ..... **vii**
List of Tables ..... **viii**
1 Introduction **1**
1.1 Context and Problem Statement 1
1.2 Contributions 2
1.3 Outline 4
2 Background and Related Work **5**
2.1 Use Cases 5
2.2 Transactions (System Transactions) 8
2.3 Business Transactions 9
2.4 Sagas 9
2.5 Concurrency Management 10
2.5.1 Pessimistic Concurrency Management 11
2.5.2 Optimistic Concurrency Management 12
2.5.3 Pessimistic and optimistic strategies: a comparison 13
2.6 Related work 14
3 Requirements Specification of Business Transactions **16**
3.1 Motivating Scenario 16
3.2 Terminology 18
3.3 Business Transaction Template 19
3.4 Use Case Example 24
3.5 Saga Template 29
3.6 Saga Example 30
3.7 Summary 38
4 Case Study **40**
4.1 Domain Model and Use Case Briefs 41
4.1.1 Domain model 41
4.1.2 Actors 42
4.1.3 Use case briefs 42
4.1.4 Non-functional requirements 43
4.2 Selected Use Cases in Detail 44
4.2.1 Use Case: Edit Student Information (optimistic) 44
4.2.2 Use Case: Import Student Registry (pessimistic) 46
4.2.3 Use Case: Delete Students from Student Registry (hybrid) 47

4.3 Summary 50
5 Writing Effective Transactions within Use Cases: Heuristic Guidelines 51
5.1 Finding Transactional Resources 51
5.2 Finding Transaction Boundaries 52
5.3 Choosing a Concurrency Management Strategy 54
5.3.1 Optimistic Concurrency Management 55
5.3.2 Pessimistic Concurrency Management 55
5.3.3 Hybrid 56
6 Conclusion and Future Work 57
References 59
Appendix A. Use Case Model: Course Management System 63
Appendix B. Domain Model 81
Appendix C. An XML UCM Meta-model for use cases with Eclipse 83

List of Figures

Figure 1. Traditional use case structure 7
Figure 2. Saga's compensation hierarchy 10
Figure 3. Example of pessimistic concurrency management 12
Figure 4. Example of optimistic concurrency management 13
Figure 5. Use case template of a business transaction 21
Figure 6. Excerpt of the domain model 25
Figure 7. Use case: add student to class list 28
Figure 8. Saga template 29
Figure 9. Domain model of trip organizer system 31
Figure 10. Saga use case example: plan and book a trip package 31
Figure 11. Use case: cancel flight reservation 35
Figure 12. Use case: cancel hotel reservation 36
Figure 13. Use case: cancel car reservation 37
Figure 14. Saga compensation diagram for saga example 38
Figure 15. Domain model of course management system 41
Figure 16. Use case: edit student information in student registry 45
Figure 17. Use case: import student registry 46
Figure 18. Use case: delete students from student registry 48
Figure 19. Main success scenario of "add student(s) to class list" without transactional boundaries 53
Figure 20. Main success scenario of "add students(s) to class list" with transactional boundaries 54
Figure 21. Example of a simple domain model 82
Figure 22. Example of a rich domain model 82
Figure 23. Structure of the package 84
Figure 24. Structure of the use case 85
Figure 25. Structure of the main success scenario 86
Figure 26. Structure of the extensions 87
Figure 27. Structure of the extension 88

List of Tables

Table 1. A comparison between pessimistic and optimistic concurrency management... 14
Table 2. Actors and their responsibilities ... 42
Table 3. Use case briefs of course management system ... 43
Table 4. List of DTD elements ... 83

1 Introduction

1.1 Context and Problem Statement

An *Enterprise Application (EA)* is a software system that implements numerous business services, which are simultaneously accessed by a large number of users. Typical examples of the kinds of services provided by EAs include high level business operations such as managing a bill of materials or production planning, to lower level functions such as retrieving a list of employees and confirming a vacation request. EAs play a crucial role in our modern economy. For example, in the third quarter of 2006 alone, US retail *e*-commerce sales totaled over 27 billion dollars [US Census 2006].

The complexities of EAs are multi-dimensional. This makes EA requirements specification and design a particularly challenging task. For example, EAs implement the business logic of the surrounding organization. The logic must follow business rules that often have evolved over years and feature a multitude of exceptional cases. Also, an EA offer users concurrent access to its services. The ability to deal with concurrent user access to data is one of the most tricky and error prone aspects of EAs [Fowler 2003]. When multiple clients attempt to access the same data concurrently special care must be taken in choosing appropriate concurrency control mechanisms and conflict resolution strategies.

To help tackle the challenging issue of concurrency management, software engineers have learned to make use of an abstraction known as *business transactions* [Fowler 2003]. A business transaction is a logical unit of interaction between two (or more)

business entities. Similar to a system transaction—i.e., an internal query to a database system—business transactions are bound to the ACID properties (Atomicity, Consistency, Isolation and Durability) [Ramakrishnan & Gehrke 2002]. Partial executions are not desirable and when they cannot be avoided, they require the execution of recovery actions. In contrast to system transactions, business transactions may stretch over a long period (hence are sometimes referred to as long-lived transactions) and the execution of their actions may depend on the interaction of associated actors.

In this thesis, we argue that functional requirements and the involved user interaction are dependent on the chosen concurrency management strategy of the encompassing business transaction. Use cases are the specification technique of choice for functional requirements documentation. A use case captures the interaction between actors and the system under development. A use case is organized as a collection of related success and failure scenarios that are all bound to the same goal of the primary actor [Larman 2005]. Unfortunately, the current state of the art in use case writing does not provide a proper means to express business transaction specifications.

The main goal of this thesis is to overcome this shortcoming by demonstrating how business transactions can be documented in use case specifications. In particular, we adapt the use case notation to allow business transactions to be defined in terms of its boundaries, affected resources and the chosen concurrency management strategy.

1.2 Contributions

The main contributions of this thesis are to:

- Demonstrate that analysis and modeling of business transactions is a domain activity.
- Present an adaptation to the use case notation for capturing business transactions.
- Validate the proposed adaptation by applying it to a comprehensive case study of a Course Management System.
- Describe a set of heuristics that will be useful in the analysis and modeling of business transactions at the requirements level.

Next, we describe each of the main contributions in greater detail.

Analysis and Modeling of business transactions is a domain activity. Unfortunately, most literature deals with business transaction modeling from a design perspective only [Ariafai et al. 2006, Butler et al. 2005, Subrahmanyam & Richard 2000] and a false impression is given in which transaction modeling is solely a responsibility of the software designer. In this thesis, we demonstrate that the choice of concurrency management strategy depends on the domain and hence should be modeled during requirements elicitation and documentation. Failing to do so can have a significant negative effect on the usability of the system.

Specifying business transaction within use cases. The state of the art in use case writing does not provide us with constructs for specifying business transactions. Therefore, we define a use case meta-model which includes both traditional use case constructs and constructs for indicating transaction boundaries, the chosen concurrency management strategy, and the involved transactional resources. In this vein, we also define how sagas (sets of business transactions with relaxed ACID properties) can be modeled within use case specifications.

Case study and heuristics for modeling business transactions. We plan to validate our research by carrying out a comprehensive case study. Experiences and lessons learnt from the case study are distilled into a set of heuristics and practical guidelines, which assist the software engineers in:

- Choosing the proper concurrency management strategy.
- Identifying transactional resources.
- Determining transaction boundaries.

In addition, we compile a list of typical use case extensions that are likely to appear for a particular concurrency management strategy.

1.3 Outline

The remainder of this thesis is structured as follows. **Chapter 2** reviews background information and relevant related work. In **Chapter 3,** we define our extension for use cases which allows modeling business transactions and sagas. **Chapter 4** presents a case study whose main goal is to investigate the practical applicability of the proposed extension. **Chapter 5** summarizes the lessons learnt from the case study and provides a set heuristics and practical guidelines. Finally, in **Chapter 6** we conclude and provide an outlook to future avenues.

2 Background and Related Work

In this chapter, we review some important definitions that are required to understand the rest of the thesis. We define use cases as a basic technique to capture the system requirements and explain its role in the software design life cycle. Then we introduce transactions, business transactions and sagas followed by the definition of different types of concurrency management. The rest of this chapter is dedicated to related work. We cover existing approaches to modeling concurrency management and transactions in the requirements phase.

2.1 Use Cases

Use cases were introduced roughly 15 years ago by Jacobson. He defined a use case as a "specific way of using the system by using some part of the functionality" [Jacobson 1992]. More recent popularization of use cases is often attributed to Cockburn [Cockburn 2001].

Use cases are typically employed as a specification technique for capturing functional requirements. Each use case consists of interactions between actors and the system under development. Actors represent users or entities (e.g. secondary systems) that interact with the system. By definition, actors are outside of the system boundary. A distinction is made between primary and secondary actors. A primary actor, typically a user, initiates the use case in order to accomplish a pre-set goal. Secondary actors support the execution of the use case through interactions with the primary actor [Gomaa 2005].

Figure 1 depicts a template for a fully dressed use case as defined by Cockburn [Cockburn 2001]. As is illustrated, every use case starts with a header section containing various properties. The "primary actor" property identifies the actor who initiates the interaction specified by the use case. The "goal" property captures the very intent the primary actor has in mind when executing the use case. "Level" indicates the goal level of the use case. While different goal levels exist, the most important ones are *summary*, *user-goal* and *sub-function*. The user-goal level use cases refer to "elementary business processes". Summary level use cases provide a context for user-goal level use cases. They also show the life cycle of related use cases and, in effect, act as a use case table of contents. User-goal use cases at times employ sub-function use cases to achieve their objectives [Cockburn 2001]. Sub-function use cases are those that are too "small" to be considered elementary business processes.

The core part of a use case is its main success scenario. It indicates the most common way in which the primary actor can reach his/her goal by using the system and interacting with secondary actors. A use case is completed by specifying use case extensions. Extensions constitute alternative scenarios, which may or may not lead to the fulfillment of the use case goal. They represent exceptional and alternative behavior (relative to the main success scenario) and are indispensable to capturing full system behavior. Each extension starts with a condition (relative to one or more steps of the main success scenario), which makes the extension relevant and causes the main scenario to "branch" to the alternative scenario. The condition is followed by a sequence of action steps, which may lead to the fulfillment or the abandonment of the use case goal and/or further

USE CASE: TITLE

PROPERTIES

- Primary Actor:
- Goal:
- Level:
- Precondition:

MAIN SUCCESS SCENARIO

step 1.
step 2...
...
step n.

EXTENSIONS

2a. Condition

- extension steps ...

2b. Condition

- extension steps ...

Figure 1. Traditional use case structure

extensions. From a requirements point of view, exhaustive modeling of use case extensions is an effective requirements elicitation device.

The complete set of use cases for a system is captured in a Use Case Model (UCM). The UCM documents the majority of software and system requirements and as such, it is the main part of the contract (of the envisioned system behavior) between stakeholders [Cockburn 2001]. Moreover, it is useful in determining the boundary of the system. Only behavior defined by the use case model is part of the envisioned system. Interactions that are not specified are deemed outside of the system's boundary [Overgaard & Palmkvist 2004]. Use case modeling is a crucial activity in the software development life cycle (e.g. Rational Unified Process) and has significant benefits for both software developers and customers to achieve a common understanding of the functionalities of the system under development [Merrick & Barrow 2005, Toerner et al. 2006, Whittle & Jayaraman 2006].

Transactions play a crucial role in software systems; next, we define transactions and their properties.

2.2 Transactions (System Transactions)

Transactions (or *system transactions*) are important concepts for handling concurrency management of enterprise applications (EAs) [Fowler 2003]. A transaction is defined as a sequence of interactions between software systems which has well-defined start and end. Every transaction must obey to the following properties (which we refer to as A.C.I.D.) [Banagala 2006]:

Atomicity. Either *all* the steps of the transaction are executed or *none*. A transaction step is said to be executed if it has a permanent (externally visible) effect on the state of any of the system's resources.

Consistency. The system must be in a consistent, non-corrupt state at the start and the finishing point of the transaction. In other words, a transaction must maintain the consistency of data in data repositories. For example, in moving money between two different accounts, the transaction must ensure that the debited amount equals the credited amount.

Isolation. The results of an individual transaction must not be visible to any other transaction until the transaction commits successfully.

Durability. This property ensures that all updates are persisted when the transaction has committed successfully. In other words, any result of a committed transaction must be made permanent.

2.3 Business Transactions

Business transactions are logical units of interaction performed between two or more business entities [Gallina & Mammar 2006]. Similar to system transactions (described in the previous section) business transaction have the ACID properties. In contrast to system transactions, business transactions are typically at a higher level of abstraction and as such that they are meaningful to users. System transactions are internal to software systems and hence essentially invisible to users. Business transactions may stretch over a long period (long running transactions) and the executions of their actions may depend on the interaction of associated actors. Examples of use cases will be given in the following chapters.

2.4 Sagas

"*Long lived transactions (LLTs) hold on to the database resources for relatively long periods of time, significantly delaying the termination of shorter and more common transactions*" [Garcia-Molina & Salem 1987]. Similar to a LLT, a saga consists of sequence of sub-transactions that can be interleaved with sub-transactions of other sagas [Garcia-Molina & Salem 1987]. Sub-transactions within a saga are related to each other and should be executed as a unit. When a saga fails, compensation actions are performed starting from the saga's failure point. The purpose of compensation actions is to undo (if possible) the effects of the transactions that have been already committed. Figure 2 shows a saga with four sub-transactions. It depicts the compensation actions to be invoked, depending on *when* the saga aborts. For example, when saga aborts during the execution of transaction D, compensation actions need to be carried out for the transactions C, B, A

(in that order). It is important to note that undoing the effects of a transaction is not always possible. In such a case, a compromise solution needs to be negotiated.

Note that a saga is like a transaction for which we have relaxed the atomicity and isolation properties of transactions, since the saga's sub-transitions are interleavable. Consequently, sub-transactions (within different sagas) can commit and roll back independently. Thus, resources may be released prior to the end of the saga, and the liveliness of the system, in terms of maximizing the degree of concurrent executions, is significantly increased [Garcia-Molina & Salem 1987].

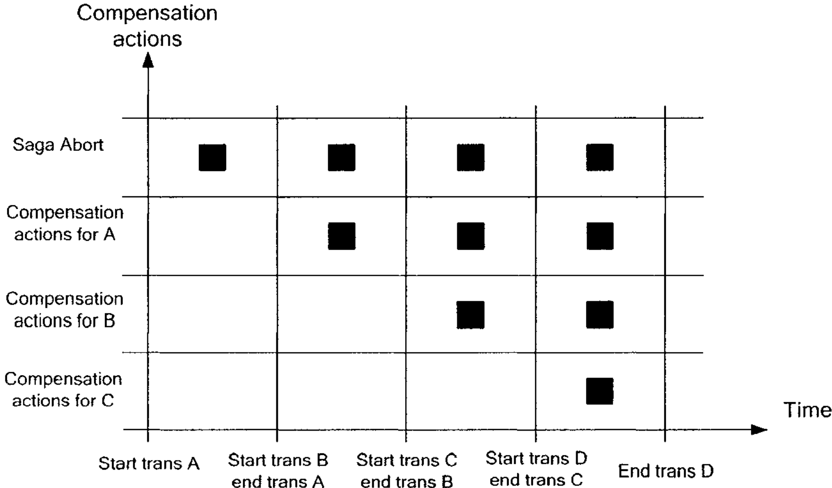


Figure 2. Saga's compensation hierarchy

2.5 Concurrency Management

One of the most complicated aspects of software development is dealing with concurrency. In order to avoid inconsistencies due to concurrency conflicts, a concurrency management strategy is needed. When modeling business transactions, the

transaction boundaries, the chosen concurrency management strategy and affected domain objects are important concepts.

A business transaction might deal with one or more domain model objects. Specifying these objects enables software engineers to determine when and which resources should be involved in the business transactions and gives a better understanding of concurrency management.

In general, we distinguish between three basic types of concurrency management: (1) pessimistic concurrency management, (2) optimistic concurrency management and (3) hybrid concurrency management.

2.5.1 Pessimistic Concurrency Management

In a pessimistic concurrency management strategy, resources are locked at the very beginning of the transaction. The main idea of pessimistic concurrency management is to prevent conflicts.

Figure 3 illustrates the interaction involved in pessimistic concurrency management in a sequence diagram. Two users (John and Daniel) intend to concurrently edit a shared resource, *R*. The transaction for John starts with the *get(R)* message being sent to the enterprise application (EA). As a result, a lock is put on resource *R*. Hence, *R* will be unavailable for other users including Daniel. The gray bar on the right hand side shows the duration of the lock on the resource *R*. John updates information in *R*. Finally, a message informs John that the update was successfully done.

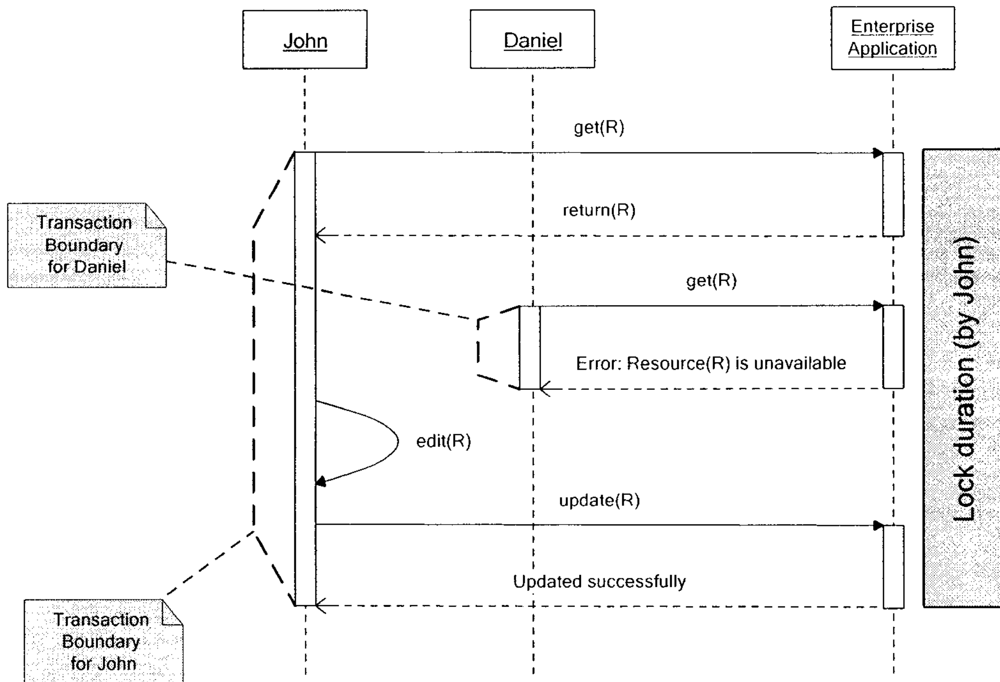


Figure 3. Example of pessimistic concurrency management

2.5.2 Optimistic Concurrency Management

In optimistic concurrency management, locking is done just before the last step of the transaction—i.e. the commit step. The main idea of optimistic concurrency management maximize system availability while ensuring that resource update conflicts are appropriately detected and resolved. Resources are only locked for the short time span of commit step of the transaction. Figure 4 provides an example of an optimistic concurrency management scenario. Similar to the previous example, we assume that there are two users (John and Daniel) who intend to concurrently edit resource *R*. The transaction for John starts with the request *get(R)*. Unlike the previous example, the system only locks *R* when John calls *update(R)*. Hence, when Daniel attempts to update

R a conflict is detected and reported. In this case, an error message informs Daniel that the information of *R* has already been changed.

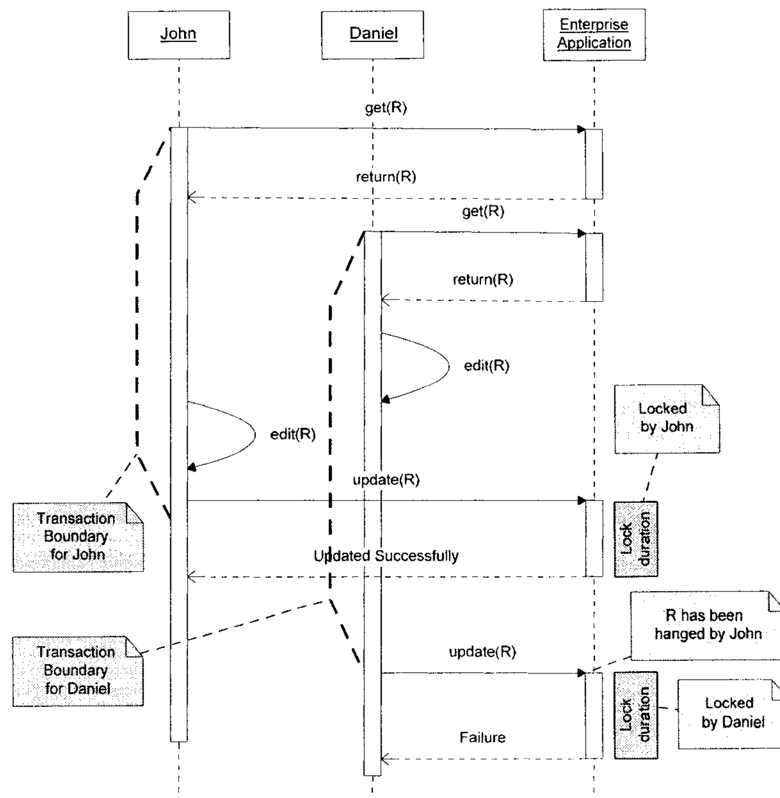


Figure 4. Example of optimistic concurrency management

Atkins and Coady define *hybrid concurrency management* as being concurrency management where locking is done neither at the beginning nor at the very end of the transaction [Atkins & Coady 1992].

2.5.3 Pessimistic and optimistic strategies: a comparison

Table 1 compares optimistic and pessimistic strategies [Atkins & Coady 1992].

Strategy / Properties	Optimistic	Pessimistic
Intention	It concerns conflict detection [Fowler 2003].	It concerns conflict prevention [Fowler 2003].
Locking duration within business transaction	Immediately before the commit step of the transaction.	Before the first step of the transaction.
Conflicts between concurrent transactions	Are identified at the end of a transaction's execution.	Not applicable. All resources are already locked. Other transactions cannot have access to the locked resources.
Efficient when	The amount of wasted work is not comparatively considerable.	The amount of wasted work is comparatively significant.
Appropriate when the chance of conflict is ...	Low	High

Table 1. A comparison between pessimistic and optimistic concurrency management

2.6 Related work

The importance of modeling transactions and concurrency management in the early development stages has been identified by various authors. Fowler emphasizes that the choice of concurrency control strategy depends to a great extent on the likelihood and the consequences of a concurrency conflict, which in turn are to be determined during requirements specification [Fowler 2003]. Banagala points out that understanding and modeling business transactions is a crucial activity during the analysis phase [Banagala 2006]. In his work, the author demonstrates how business transactions can be modeled during the analysis stage by proposing an enhancement to Jackson's problem frames approach [Jackson 2001]. Bennett *et al.* propose a framework for the analysis and the design of long running business processes and their entailed business transactions [Bennett et al. 2000]. A business process is modeled by a set of interrelated tasks resulting in a task dependency graph. During the design phase, one or more tasks are

mapped to a node in the *Unit of Work* tree, where each *Unit of Work* represents a transaction. Unfortunately, the proposed framework is not very well integrated with standard functional requirements documentation techniques such as use cases.

Within the context of transactional business software, Correa and Werner assert that scenario specifications (in form of use cases) and the specification of business rules and transactions are closely related [Correa & Werner 2004]. As such, an in-depth understanding of the underlying transaction is indispensable for the specification of interaction details. Based on this assertion, the authors propose an approach where preliminary use case specifications are enriched with OCL (Object Constraint Language) annotations. These OCL annotations specify business rules as well as pre- and post conditions of business transactions.

In context of web services development, Alrifai *et al.* propose a web-service transaction protocol, based on transaction dependency graphs [Alrifai *et al.* 2006]. The protocol ensures the consistency of data while different business transactions are dealing with the same data concurrently.

In this thesis, similar to the approach of Correa and Werner [Correa & Werner 2004], an extension for use case models is defined. In contrast to their work, however, our approach is not based on an external notation but is integrated with the use case model. Additionally, our approach closely incorporates references to domain objects and supports modeling of transactional resources.

3 Requirements Specification of Business Transactions

In this section, we propose a means of integrating business transaction requirements within use cases. First, a motivating example is presented. Next, we define a new use case template whose use can help in the specification of business transactions within use cases. Finally, we propose a similar template for writing sagas, followed by an example.

3.1 Motivating Scenario

A Flight Reservation System is a common web-based enterprise application. In the motivating example presented below, we describe an application that deals with several flight companies, which have various classes of flight with different prices. Users employ the system to book their flights concurrently. The number of empty seats in a flight is a dependant variable of the plane type and the flight class. In what follows, a scenario describes a situation in which two different users employ the system to book a seat in a same flight:

Using the flight reservation system "Get-You-There", a customer named Paul wants to book a flight from Montreal to Vancouver. After entering his flight criteria, the system suggests a set of suitable flights. Paul selects a flight and, out of excitement for having obtained a great air fare, runs into the kitchen to tell his wife about his good fortune. In the mean time, Frank, who is also interested in flying from Montreal to Vancouver, coincidentally selects the same flight as Paul. Frank lives alone and, without any delays, proceeds to select an appropriate seat and purchases the flight using his credit card. After successful validation of his credit limit, the system issues an electronic ticket. While

Frank already enjoys a cool beverage to celebrate his upcoming vacation, Paul returns to his computer in order to finalize his booking. He selects a seat and submits payment information. Unfortunately, in Paul's case, the system indicates that the selected flight in the desired booking class is already fully booked. As it turns out, Frank's booking preempted Paul's and in doing so, Frank booked the last available seat in the booking class. Paul, frustrated, returned to the kitchen to tell his wife that he will never use the system again.

Apparently, the business transaction was implemented using an optimistic concurrency management strategy, and hence, no lock was set on the corresponding flight data upon initiation of a business transaction. As a result, the system gave the impression to all interested customers that a sufficient number of seats were available, even though the number of potential customers was greater than the number of available seats. If the system had been designed using a pessimistic strategy, Paul's disappointment would have been avoided, as all resources would have been locked in Paul's favor until the business transaction had finished. Hence, Frank would not have had the chance to book the same flight. Which approach is best? It is up to all stakeholders, especially, targeted end users to decide; e.g. if the full consequences of either strategy were known to end users, they might in fact propose a third alternative where customers are aware of the number of remaining seats as well as the number of customers in contention for them.

Therefore, managing concurrency issues in the requirements phase can reduce the effort to deal with them in the design and implementation phases by placing the effort where it can yield the most benefit—i.e. as early as possible.

The main benefits of modeling business transactions, sagas and associated concurrency management can be summarized as follows:

- **Domain specific**. In our opinion, concurrency management is, to a great extent, a domain issue and hence should be analyzed and documented during requirements activities. In contrast to low-level system transactions, the nature of business transactions, sagas and their corresponding compensation actions depend on the **application domain** and the needs of stakeholders.
- **Elicitation of requirements**. Functional requirements and the involved user interaction are dependent on the chosen concurrency management strategy. For example, the functional requirements for optimistic concurrency management focus on conflict resolution whereas the functional requirements for pessimistic concurrency management are based on conflict prevention. Which strategy is chosen will affect how users will be able to interact with the system.

The main goal of this thesis is to provide support for modeling of business transactions and sagas at the requirements level, specifically through use cases. In particular, we define an **extension** to use cases to support the specification of business transactions and sagas as well as the corresponding concurrency management strategy. As discussed in the previous chapter use cases are the vehicle of choice for documenting functional requirements, which in turn are closely related to the chosen concurrency mechanism.

3.2 Terminology

The main terms used in our extension to use cases are defined as follows:

- **Transaction** denotes a use case step, whose substeps are the activities of a business transaction. Each such transaction is attributed a set of transactional resources.
- **Transactional Resource (TRs)** denotes a domain entity that is directly or indirectly either created, read, modified or deleted by a use case step within the scope of a transaction. It is important to note that since we are dealing with requirements, a transactional resource refers to a domain concept and not to a design or implementation level class.
- **Resource Lock**. Before a transactional resource can be altered, the system needs to gain exclusive access to it. This is achieved by means of a resource lock. In this thesis, we employ exclusive read/write lock mechanism to lock transactional resources.
- **Commit** and **Abort**. The last substep of a transaction is a commit. In committing a transaction, its effects become permanent and visible to others. For various reasons (some of which will be explored later), a transaction can be aborted in which case any intermediate substeps are "undone" so that the system is left with no trace that the transaction was ever initiated. An aborted transaction has no net effect. In our approach, the main success scenario of the use case contains a COMMIT step and a TRANSACTION ABORT can only occur in the corresponding use case extensions; both lead to the release of the resource locks.

3.3 Business Transaction Template

In this section, we define a "template" for the specification of business transactions inside use cases. As depicted in Figure 5, business transaction modeling is captured in the main success scenario as well as in use case extensions. Note that for sake of readability, we

have collapsed sequences of zero or more steps (written as step*) whose details are not relevant to our presentation into a single numbered step—e.g. steps 1 and 3 of the main success scenario.

USE CASE:

PROPERTIES:

- Primary actor:
- Goal:
- Level:
- Minimal Guarantee:
- Frame:
- Precondition:

MAIN SUCCESS SCENARIO

1. step*.
2. TRANSACTION.

   Transactional Resources (TRs): list of TRs.

   2.1 step* (possibly accessing TRs).

   2.2 BEGIN RESOURCE LOCK of type: [*exclusive read* | *exclusive write* | *read/write*].

   2.3 step* (possibly accessing, changing TRs).

   2.4 COMMIT.
3. step*.

EXTENSIONS

(2.1-2.3)Primary actor indicates that he/she wishes to abort the transaction:

- step*.
- TRANSACTION ABORT.
- step*.

(2.1-2.4) System detects that Transactional Resources have been changed by another transaction:

- step* (conflict resolution).
- TRANSACTION ABORT (*optional*).
- step* (resume in Main Success Scenario if transaction was not aborted).

(2.2) Transactional Resources are already locked:

- step* (conflict resolution).
- TRANSACTION ABORT (*optional*).
- step* (resume in Main Success Scenario if transaction was not aborted).

(2.3) Response timeout: *(optional: applicable in pessimistic/hybrid concurrency management)*

- step*.
- TRANSACTION ABORT.
- step*.

(2.4) Failure persisting data:

- step* (*recovery*).
- TRANSACTION ABORT.
- step*.

Figure 5. Use case template of a business transaction

Main Success Scenario. The start of a transaction is indicated by the keyword TRANSACTION followed by the list of transactional resources used in the transaction. The main success scenario can have only one transaction and does not support nested transactions (a main success scenario containing more than one transaction is called a saga—sagas are covered in Section 2.4). The transaction will contain a sequence of substeps consisting of user interactions and internal system computations including accessing and modifying the transactional resources. Note that modification of transactional resources remain invisible to others (by nature of the isolation principle of transactions). Two internal system steps are present in every business transaction: the first is denoted by BEGIN RESOURCE LOCK and it indicates the obtaining of exclusive access to the transactional resources; the second, a COMMIT, marks the successful termination of the transaction. Note that depending on *when* the transactional resources are locked, either an optimistic, a pessimistic, or a hybrid concurrency management strategy is defined. Specifically, we have a

- *Pessimistic strategy* when there are no steps in 2.1; i.e. the transaction actually starts with BEGIN RESOURCE LOCK.
- *Optimistic strategy* when there are no steps between the lock and the commit (i.e. 2.3 is empty).
- Hybrid strategy otherwise.

While general guidelines are applicable, the decision as to the most appropriate choice of locking strategy can be made by domain experts [Fowler 2003]. Domain experts are also those who are in the best position to decide what should be done during transaction

conflicts or failures as well as the most suitable resolution and recovery actions—we discuss these next.

Extensions. As depicted in the template (Figure 5), there also exists a set of extensions associated with a business transaction. As will be clarified below, not all extensions shown in the template are applicable in all cases.

(2.1-2.3) Primary actor indicates that he/she wishes to abort the transaction. At any time between the start of the transaction and the COMMIT step, the user may request that the transaction be aborted. The extension steps that follow a request for abort generally consist of the system asking for confirmation, the user acknowledging, and the transaction being aborted—of course, more complex interactions are possible. As was mentioned earlier, the TRANSACTION ABORT causes the transactional resources to be released and to be left unchanged.

(2.1-2.4) System detects that Transactional Resources have been changed by another transaction. When locking is not pessimistic (i.e. there are one or more steps in 2.1), transactional resources may be changed concurrently by multiple transactions due to the absence of a resource lock during the step(s) of 2.1.

In order to avoid lost updates and inconsistent reads [Fowler 2003], the system must ensure that transactional resources have not been updated by another transaction prior to the COMMIT step. To offer maximum flexibility to designers, the system is given the freedom to report the detection of changed resources at any point from 2.1 to 2.4. When a conflict is detected, conflict resolution measures are taken: in the simplest case, this involves notifying the user and aborting the transaction. Making the right choices with

respect to conflict resolution is a good example of key opportunities for requirements elicitation.

(2.2) Transactional Resources are already locked. This extension occurs if the system fails to obtain a resource lock because one or more of the transactional resources have already been locked by another transaction. Under such circumstances the transactional resources cannot be modified and the transaction must be aborted or the primary actor given the opportunity to indicate that a resource lock be reattempted.

(2.3) Response time-out. A common transaction failure scenario pertinent to the pessimistic concurrency management strategy is a response time-out. Such an extension is necessary to ensure that resources are not locked indefinitely, should the user be unable to complete the transaction—e.g. the client is operating from a remote site and their network connection to the application server is lost.

(2.4) Failure persisting data. During the commit step of the transaction it is possible that the system fails to persist the changes made to the transactional resources, hence recovery actions become necessary. These actions may either lead to an attempted re-execution of the commit step or to the transaction being aborted.

3.4 Use Case Example

An example of a use case, which follows our template, is given in Figure 7. This "Add Student to Class List" use case is an excerpt from an ongoing research project (see appendix A) in which we are developing a *Course Manager* application. We have modified the use case step numbering to conform to the template so as to make reference

to the template easier. Moreover, for the sake of brevity, only the main success scenario and the extensions are shown in full, the header section of the use case is partly omitted.

As is usual, the use case has been carefully written to exclude user interface details [Cockburn 2001, Lilly 1999]. That is, the various interaction steps are described at high enough a level of abstraction to be applicable to several kinds of user interfaces, any of which could be adopted for the system.

An excerpt from the system Domain Model showing the concepts relevant to our example use case is given in Figure 6. From this diagram we can learn that there is a single application-wide Student Registry associated with all Students—hence, in particular, there exists no student who is not part of the Student Registry. A Student can be a member of one or more Class Lists and each Class List is associated with the offering of a course for a particular term.

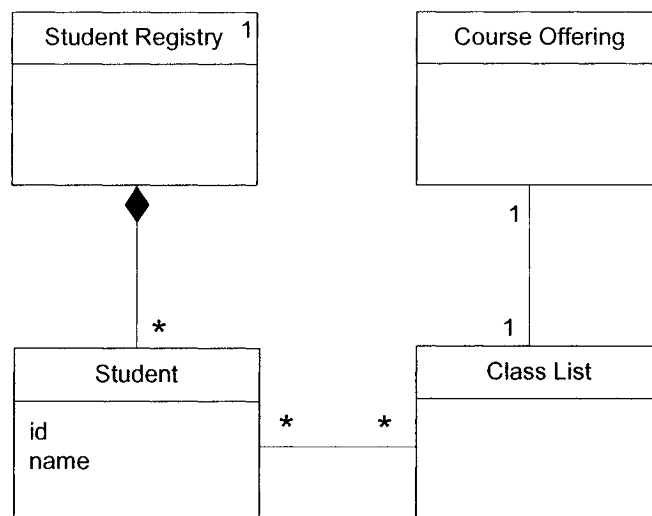


Figure 6. Excerpt of the domain model

In the use case, the trigger for the main success scenario is the primary actor's request to add a student to a class list. The next step defines the business transaction for the use case and declares the Student Registry and all Class Lists as transactional resources. As a first transaction substep the primary actor identifies a Class List and the Student to be added. This is followed by an input validation step performed by the system. Note that the use case does not constrain how the Class List and Student are identified since the exact mechanism by which this will be done (e.g. free form data entry on the course title, use of simple pattern matching on the student name, or fixed-content pull down lists) will be determined during user interface design. While for some UIs input validation may be trivial (e.g. when data selection is from a fixed-content pull down list[1]), it is a step which is generally necessary. Failure in validating the input results in extensions **(3.1a)** or **(3.1b)**, both of which allow the user to repeat the input step. Note that in step 3.1 we use the alias <Alice> to represent the student to be added; we find that this makes the use case more readable and less verbose than "the student to be added".

Next, the system attempts to lock the transactional resources. The situation where transactional resources are already locked is covered in extension **(3.2)**, which, upon acknowledgement of the primary actor, will return to step 3.2 in the main success scenario to re-attempt to gain a lock. After obtaining the resource lock, the system adds <Alice> to the Class List. On a successful commit, the system notifies the primary actor of the success of the transaction and this marks the end of the main success scenario.

A failure in persisting the data—e.g. a remote database becomes inaccessible—during the commit step leads to an extension of step **(3.4)** which aborts the transaction and ends the

[1] Even in the case of fixed-content pull down list validation is necessary if the enterprise application is made accessible by means of the web.

use case—of course other more complex recovery schemes are possible but since such events are deemed very unlikely, our efforts were better spent refining other more likely scenarios.

At any time during the transaction the system may detect a concurrency conflict in which a transactional resource has been changed by another transaction. In such a situation (extension **3.1-3.4**) the conflict is also resolved by notifying the primary actor and giving him/her a chance to try again as of step 3.1.

Finally, we point out that the example use case makes adopts an optimistic concurrency management strategy. As such, the transactional resources are not locked immediately as of the start of the transaction. Instead, despite the possibility of a concurrency conflict (extension **3.1-3.4**), the primary actor is prompted to identify <Alice> and a Class List. As a direct consequence, there is no need for our sample use case to have a time-out extension. The transactional resources are only locked for a minimal period of time, which leaves essentially no room for user interaction—hence, a time-out is unnecessary. The main reason for the adoption of optimistic concurrency management is the low probability of a conflict and the relatively low cost in its resolution—the primary actor merely repeats use case step 3.1.

USE CASE: ADD A STUDENT TO CLASS LIST

Primary Actor: Instructor. [Other use case properties omitted.]

Frame: Student Registry, all Class Lists.

MAIN SUCCESS SCENARIO

1. (Trigger) Primary actor indicates that he/she wishes to add a Student from the Student Registry to a Class List.
2. System acknowledges the primary actor's request.
3. TRANSACTION. TRANSACTIONAL RESOURCES: the Student Registry and all Class Lists.
   3.1 At the System's prompt, the Primary actor identifies a Class List and the student to be added (whom we will call <**Alice**>); System ensures that the provided information is valid by: ... [validation details omitted] .
   3.2 BEGIN RESOURCE LOCK: exclusive read/write.
   3.3 System adds <Alice> to the Class List.
   3.4 COMMIT.
4. System notifies primary actor that <Alice> has been successfully added.

EXTENSIONS

(3.1-3.3) Primary actor indicates that he/she wishes to abort the transaction:

- TRANSACTION ABORT.
- Use case ends unsuccessfully. Transactional resources remain unchanged.

(3.1-3.4) System detects that Transactional Resources have been changed by another transaction:

- System notifies primary actor that TRs have been changed by another transaction.
- Use case resumes at 3.1.

(3.1a) The id given for <Alice> is not in the Student Registry:

- System informs primary actor that information of <Alice> is invalid—e.g. invalid student id.
- Use case resumes at 3.1.

(3.1b) <Alice> is already in the Class List:

- System informs primary actor that <Alice> is already in the Class List.
- Use case resumes at 3.1.

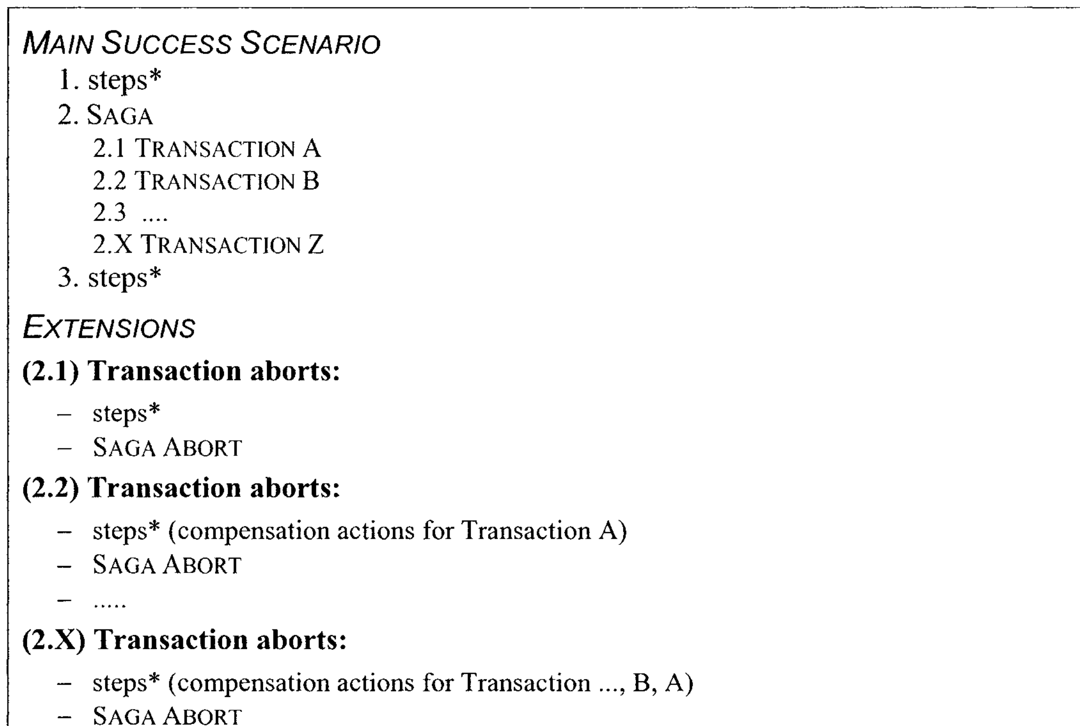
(3.2) Transactional Resources are already locked:

- System notifies primary actor that TRs are already locked by another transaction.
- System asks the primary actor if he/she wishes the system to retry obtaining a lock.
- Primary actor acknowledges. [Extension to this step is not shown.]
- Use case resumes at 3.2.

(3.4) Failure persisting data:

- System notifies primary actor that the transaction could not be completed because the changes could not be persisted.
- TRANSACTION ABORT.
- Use case ends unsuccessfully. Transactional resources remain unchanged.

Figure 7. Use case: add student to class list



MAIN SUCCESS SCENARIO
1. steps*
2. SAGA
   2.1 TRANSACTION A
   2.2 TRANSACTION B
   2.3 ....
   2.X TRANSACTION Z
3. steps*

EXTENSIONS

(2.1) Transaction aborts:
- steps*
- SAGA ABORT

(2.2) Transaction aborts:
- steps* (compensation actions for Transaction A)
- SAGA ABORT
- .....

(2.X) Transaction aborts:
- steps* (compensation actions for Transaction ..., B, A)
- SAGA ABORT

Figure 8. Saga template

3.5 Saga Template

A saga consists of a sequence of two or more subordinate transactions. Sub-transactions within a saga are related to each other and should be executed as a unit. In the case of a partial execution, *compensation actions* are invoked starting from the point where the saga failed. In what follows, we define a template for representing sagas in use cases. As depicted in Figure 8, a saga specification is captured in the main success scenario as well as in use case extensions.

Main Success Scenario. The start of a saga is indicated by the keyword SAGA. Within a SAGA step, a sequence of transactions is given. Each transaction is defined according to the template given in Figure 5, including all relevant extensions.

Extensions. The main issue to be dealt with in the extensions of a saga is the compensation actions necessary for any previously committed transaction. A compensation action tries to undo the changes in order to recover the state prior execution of the saga. In some circumstances, it may not be possible to undo all the changes. In this case, the compensation action attempts to offer the primary actor some form of reimbursement for the loss of time, money or effort due to the failure to complete the saga. As illustrated, compensation actions are invoked in reverse order of their transaction counterparts, starting from the point where the transaction failed.

3.6 Saga Example

In what follows, we propose a "Trip Organizer System" as a means of illustrating a saga. Suppose the system allows users to organize their business or personal trips by choosing a travel package. We assume that users interact with the system online by specifying the origin, destination, date and the period of their trip. In response, system presents a collection of packages for the trip. Each package consists of a flight, a hotel and rental car. The payment method is by credit card. The system also creates an Account (initially with zero balance) that will be used when a user wants to cancel a travel package. User has an option to keep the travel package price as a credit for his/her future travels. In addition, system can use the account for offering promotions to its clients. Figure 9 illustrates the domain model of the "trip organizer system".

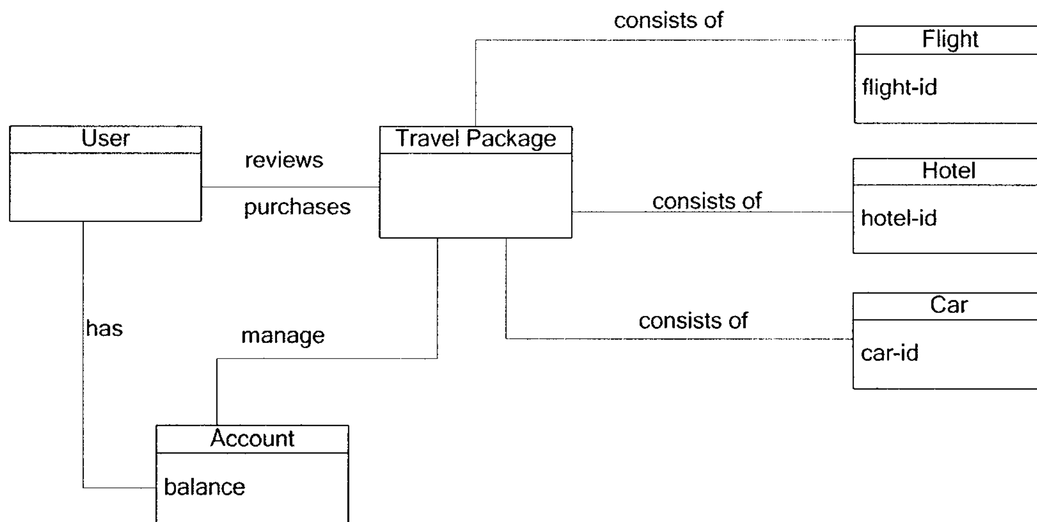


Figure 9. Domain model of trip organizer system

In what follows we present a set of use cases, which capture the before-mentioned saga.

Figure 10. Saga use case example: plan and book a trip package

USE CASE: PLAN AND BOOK A TRIP PACKAGE

Primary Actor: User of the system who wants to organize a trip
Goal: Primary actor successfully purchases a travel package including a flight, hotel reservation and car reservation by using the system.
Level: User Goal.
Minimal Guarantee: No action, no purchases.
Frame: Account, Flight, Hotel, Car.
Precondition:

- Primary Actor is authenticated

MAIN SUCCESS SCENARIO

1. Primary Actor indicates that he/she wishes to plan and book a trip by using the system.
2. System acknowledges the Primary Actor's request.
3. SAGA
   3.1 TRANSACTION: Reserve Flight. TRANSACTIONAL RESOURCES: Flight, Account.
      3.1.1 In response to a system prompt, the Primary actor provides information needed for the flight reservation including origin, destination, dates, name of passenger(s) and flight class.
      3.1.2 System provides list of available flights and their cost.
      3.1.3 Primary actor indicates his/her choice of flight.
      3.1.4 Primary actor provides his/her credit card information.

3.1.5 BEGIN RESOURCE LOCK: exclusive read/write.
3.1.6 System validates the credit card information and deducts the corresponding amount from the Account (up to a balance of 0) and applies remaining charges to the credit card.
3.1.7 System reserves the flight.
3.1.8 COMMIT.

3.2 TRANSACTION: Reserve Hotel. TRANSACTIONAL RESOURCES: Hotel, Account.
3.2.1 In response to a system prompt, the Primary Actor provides information needed for the hotel reservation including, hotel type, number of guests and number of rooms.
3.2.2 System provides a list of available hotels at the destination along with their room rates.
3.2.3 Primary Actor selects hotel(s) and room(s) for the period of time that he/she wishes to reserve.
3.2.4 BEGIN RESOURCE LOCK: exclusive read/write.
3.2.5 System validates the credit information and deducts the corresponding amount from the Account.
3.2.6 System reserves the hotel room(s).
3.2.7 COMMIT.

3.3 TRANSACTION: Reserve Car. TRANSACTIONAL RESOURCES: Car, Account.
3.3.1 In response to a system prompt, the Primary Actor provides the information needed for the car reservation including the type of car, and the rental period.
3.3.2 System provides the list of car rental companies and their available cars for the given period.
3.3.3 Primary Actor selects a car from the given list.
3.3.4 BEGIN RESOURCE LOCK: exclusive read/write.
3.3.5 System validates the credit information and deducts the corresponding amount from the Account.
3.3.6 System reserves the chosen car for the specified period.
3.3.7 COMMIT.

4. System notifies the Primary Actor that the travel package has been approved and issues the tickets and necessary documents.

EXTENSIONS

(3.1.1-3.1.5) a. Primary Actor indicates that he/she wishes to abort the saga:

- TRANSACTION ABORT (RESERVE FLIGHT).
- SAGA ABORT.
- Use case ends.

(3.1.2) a. No flight is available for the specified period:

- System notifies the Primary Actor that the there is no flight available for the specific period.
- Use case resumes at 3.1.1.

(3.1.3) a. Primary Actor indicates that he/she wishes to change the flight reservation parameters (e.g. date, flight class):

- Use case resumes at 3.1.1.

(3.1.5) a. Transactional resources are already locked:

- 1. System notifies the Primary Actor that TRs are already locked by another transaction.

- 2. System asks the Primary Actor if he/she wishes the system to retry obtaining a lock.
- 3. Primary Actor acknowledges.
- 4. Use case resumes at 3.1.5.

(3.1.8) a. Failure persisting data:

- System notifies the Primary Actor that the transaction could not be completed because the change could not be persisted.

(3.1.3-3.1.5) a. System detects that the flight is already booked by another transaction:

- System notifies the Primary Actor that the flight is booked by another transaction and requests that another flight be chosen.
- Use case resumes at 3.1.2.

(3.1.5a.3) a. Response time-out:

- TRANSACTION ABORT (RESERVE FLIGHT).
- SAGA ABORT.
- Use case ends.

(3.2.1-3.2.4) a. Primary Actor indicates that he/she wishes to abort the saga:

- TRANSACTION ABORT (RESERVE HOTEL).
- CANCEL FLIGHT RESERVATION.
- SAGA ABORT.
- Use case ends.

(3.2.2) a. No rooms are available for the specified period:

- System notifies the Primary Actor that the hotel does not have available rooms.
- Use case resumes at 3.2.1.

(3.2.3) a. Primary Actor indicates that he/she wishes to change the hotel reservation parameters:

- Use case resumes at 3.2.1.

(3.2.4) a. Transactional resources are already locked:

- 1. System notifies the Primary Actor that TRs are already locked by another transaction.
- 2. System asks Primary Actor if he/she wishes system to retry obtaining a lock.
- 3. Primary Actor acknowledges.
- 4. Use case resumes at 3.2.4.

(3.2.4a.3) a. Response time-out:

- TRANSACTION ABORT (RESERVE HOTEL).
- CANCEL FLIGHT RESERVATION.
- SAGA ABORT.
- Use case ends.

(3.2.7) a. Failure persisting data:

- System notifies the Primary Actor that the transaction could not be completed because the change could not be persisted.
- TRANSACTION ABORT (RESERVE HOTEL).
- CANCEL FLIGHT RESERVATION.
- SAGA ABORT.
- Use case ends.

(3.3.1-3.3.4) a. Primary Actor indicates that he/she wishes to abort the saga:

- TRANSACTION ABORT (RESERVE CAR).
- CANCEL HOTEL RESERVATION.
- CANCEL FLIGHT RESERVATION.
- SAGA ABORT.
- Use case ends.

(3.3.2) a. No car rentals match the given reservation parameters:

- System notifies the Primary Actor.
- Use case resumes at 3.3.1.

(3.3.3) a. Primary Actor indicates that he/she wishes to change the car reservation parameters:

- Use case resumes at 3.3.1.

(3.3.4) a. Transactional resources are already locked:

- 1. System notifies the Primary Actor that TRs are already locked by another transaction.
- 2. System asks Primary Actor if he/she wishes system to retry obtaining a lock.
- 3. Primary Actor acknowledges.
- 4. Use case resumes at 3.3.4.

(3.3.4a.3) a. Response time-out:

- TRANSACTION ABORT (RESERVE CAR).
- CANCEL HOTEL RESERVATION.
- CANCEL FLIGHT RESERVATION.
- SAGA ABORT.
- Use case ends.

(3.3.7) a. Failure persisting data:

- System notifies the Primary Actor that the transaction could not be completed because the change could not be persisted.
- TRANSACTION ABORT (RESERVE CAR).
- CANCEL HOTEL RESERVATION.
- CANCEL FLIGHT RESERVATION.
- SAGA ABORT.
- Use case ends.

In the Figure 10, when the saga is aborted, compensation actions are performed which cancel the bookings. The compensation actions are captured in their own uses cases—these are described next.

USE CASE: CANCEL FLIGHT RESERVATION

Primary Actor: User of the system who wants to organize a trip

Goal: Primary actor successfully cancels the flight reservation.

Level: subfunction

Minimal Guarantee: No action, no flight cancellation.

Frame: Account, Flight, Payment list.

Precondition:

- Primary Actor is authenticated

MAIN SUCCESS SCENARIO

1. TRANSACTION: CANCEL FLIGHT RESERVATION
   TRANSACTIONAL RESOURCES: Flight, Payment list, Account.
   1.1 System informs the Primary Actor that he/she has the option to either (i) keep the full payment as a credit for future use or (ii) get a refund with a 5% cancellation fee.
   1.2 Primary Actor indicates that he/she wishes option (ii).
   1.3 BEGIN RESOURCE LOCK: exclusive read/write.
   1.4 System cancels the flight reservation, and credits the Primary Actor's credit card for the full amount less 5%.
   1.5 COMMIT.
2. System notifies the Primary Actor that the flight booking has been successfully canceled and credited according to the chosen option.

EXTENSIONS

(1.1-1.3) a. Primary Actor indicates that he/she wishes to abort the transaction:

- Use case ends unsuccessfully. Transactional resources remain unchanged.

(1.2) a. Primary Actor indicates that he/she wishes option (i):

- 1. BEGIN RESOURCE LOCK: exclusive read/write.
- 2. System cancels the flight reservation.
- 3. System transfers an amount equivalent to the price of the flight booking to the Account of the Primary Actor.
- 4. Use case resumes at step 1.5.

(1.2a.1) a. Transactional resources are already locked:

- System logs the error and marks it to be dispatched manually.
- System suspends the Account of the Primary Actor.
- System notifies the Primary Actor that an error occurred and that his/her Account is temporarily suspended and that he/she should notify customer service.
- TRANSACTION ABORT .
- Use case ends unsuccessfully. Transactional resources remain unchanged.

(1.3) a. Transactional resources are already locked:

- 1. System notifies that TRs are already locked by another transaction.
- 2. System asks the Primary Actor if he/she wishes the system to retry obtaining a lock.
- 3. Primary Actor acknowledges.
- 4. Use case resumes at 1.3.

(1.3a.3) a. Response time-out:

- TRANSACTION ABORT. Use case ends unsuccessfully. TRs remain unchanged.

Figure 11. Use case: cancel flight reservation

USE CASE: CANCEL HOTEL RESERVATION

Primary Actor: User of the system who wants to organize a trip
Goal: Primary actor successfully cancels the hotel reservation.
Level: subfunction
Minimal Guarantee: No action, no hotel cancellation.
Frame: Account, Hotel, Payment list.
Precondition:

- Primary Actor is authenticated

MAIN SUCCESS SCENARIO

1. TRANSACTION: CANCEL HOTEL RESERVATION
   TRANSACTIONAL RESOURCES: Hotel, Payment list, Account
   1.1 System informs the Primary Actor that he/she has the option to either (i) keep the full payment as a credit for future use or (ii) get a refund with a 5% cancellation fee.
   1.2 Primary Actor indicates that he/she wishes option (ii).
   1.3 BEGIN RESOURCE LOCK: exclusive read/write.
   1.4 System cancels the hotel reservation, and credits the Primary Actor's credit card for the full amount less 5%.
   1.5 COMMIT.
2. System notifies the Primary Actor that the hotel has been successfully canceled.

EXTENSIONS

(1.2) a. Primary Actor indicates that he/she wishes option (i):

- 1. BEGIN RESOURCE LOCK: exclusive read/write.
- 2. System cancels the hotel reservation.
- 3. System transfers adds the payment to the Account of the Customer
- 4. Use case resume as step 1.5.

(1.2a.1) a. Transactional resources are already locked:

- System logs the error and marks it to be dispatched manually.
- System suspends the Account of the Primary Actor, notifies that an error occurred and his/her Account is temporarily suspended.
- TRANSACTION ABORT.
- Use case ends unsuccessfully. Transactional resources remain unchanged.

(1.1-1.3) a. Primary Actor indicates that he/she wishes to abort transaction:

- TRANSACTION ABORT.
- Use case ends unsuccessfully. Transactional resources remain unchanged.

(1.3) a. Transactional resources are already locked:

- 1. System notifies that TRs are already locked by another transaction.
- 2. System asks the Primary Actor if he/she wishes the system to retry obtaining a lock.
- 3. Primary Actor acknowledges.
- 4. Use case resumes at 1.3.

(1.3a.3) a. Response time-out:

- TRANSACTION ABORT.
- Use case ends unsuccessfully. Transactional resources remain unchanged.

Figure 12. Use case: cancel hotel reservation

USE CASE: CANCEL CAR RESERVATION

Primary Actor: User of the system who wants to organize a trip
Goal: Cancel a car reservation and gets a refund with 5% cancellation fee.
Level: subfunction
Minimal Guarantee: No action, no car cancellation.
Frame: Account, Car, Payment list.
Precondition: Primary Actor is authenticated.

MAIN SUCCESS SCENARIO

1. TRANSACTION: CANCEL CAR RESERVATION
   TRANSACTIONAL RESOURCES: Car, Payment list, Account
   1.1 System informs the Primary Actor that he/she has the option to either (i) keep the full payment as a credit for future use or (ii) get a refund with a 5% cancellation fee.
   1.2 Primary Actor indicates that he/she wishes option (ii).
   1.3 BEGIN RESOURCE LOCK: exclusive read/write.
   1.4 System cancels the car reservation, and credits the Primary Actor's credit card for the full amount less 5%.
   1.5 COMMIT.
2. System notifies the Primary Actor that the car has been successfully canceled.

EXTENSIONS

(1.2) a. Primary Actor decides to keep the payment as a credit for future use:

- 1. BEGIN RESOURCE LOCK: exclusive read/write.
- 2. System cancels the car reservation.
- 3. System transfers adds the payment to the Account of the Customer
- 4. COMMIT.
- 5. System notifies the Primary Actor that the car reservation fee is kept as a credit in his/her Account.

(1.2a.1) a. Transactional resources are already locked:

- System logs the error and marks it to be dispatched manually.
- System suspends the Account of the Primary Actor.
- System notifies the Primary Actor that an error occurred and that his/her Account is temporarily suspended.
- TRANSACTION ABORT. Use case ends unsuccessfully. TRs remain unchanged.

(1.1-1.3) a. Primary Actor indicates that he/she wishes to abort transaction:

- TRANSACTION ABORT. Use case ends unsuccessfully. TRs remain unchanged.

(1.3) a. Transactional resources are already locked:

- 1. System notifies that TRs are already locked by another transaction.
- 2. System asks the Primary Actor if he/she wishes the system to retry obtaining a lock.
- 3. Primary Actor acknowledges.
- 4. Use case resumes at 1.3.

(1.3a.3) a. Response time-out:

- TRANSACTION ABORT. Use case ends unsuccessfully. TRs remain unchanged.

Figure 13. Use case: cancel car reservation

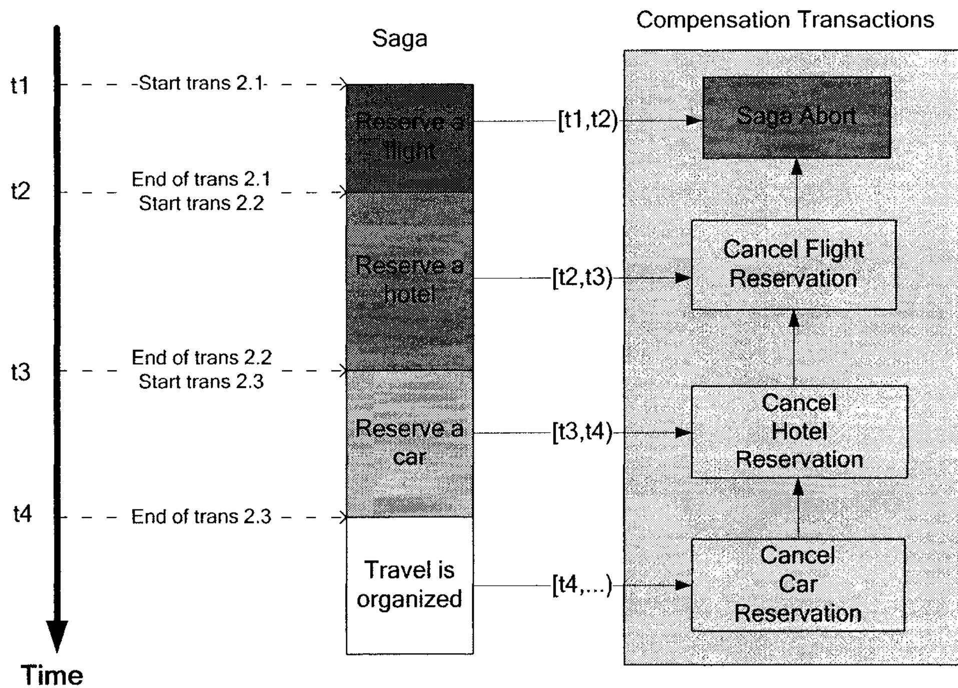


Figure 14. Saga compensation diagram for saga example

Figure 14 graphically illustrates the compensation activities related to the transactions of the saga in the given example. The arrows from the Saga to the compensation actions are adorned with expressions of the form $[t_m,t_n)$ showing the relevant time interval during which the corresponding compensation action would be triggered (i.e., the time between t_m and t_n, including t_m but excluding t_n).

3.7 Summary

We have presented an extension for use cases to capture business transaction requirements as well as sagas. The challenge at hand was to define a well-integrated extension that preserves the intuitive nature of use cases. As a first step, we established a vocabulary that could be easily understood by most stakeholders. The vocabulary

captures all relevant core concepts for modeling business transactions at the requirements level.

4 Case Study

In this chapter, we introduce the case study of a "Course Management System". The system is a web-based application that can be employed in universities to manage courses, classes and assignments. The major features of the course management system are as the follows:

- Register and manage students in the Student Registry.
- Define and manage courses and classes.
- Assign students to classes.
- Create and manage groups of students within classes.
- Submission of individual/group assignments.
- Submission of assignments grades and sending feedback to students.

The purpose of the case study is to demonstrate how the templates introduced in Chapter 3 are used to specify business transactions. In particular, we wish to demonstrate that software engineers will be able to write pessimistic, optimistic, and hybrid concurrency management strategies with the provided use case templates. We start by providing a domain model, an actor goal list and set of use case briefs. We then demonstrate (using a set of selected use cases) how business transactions are captured in use cases.

4.1 Domain Model and Use Case Briefs

4.1.1 Domain model

Figure 15 illustrates the domain model of the Course management system, portraying relevant entities of the problem domain. As can be seen, a Registrar Employee manages the Student Registry which consists of a set of Students. There is only one Student Registry. Students who are associated to particular Course Offering form a Class List. Courses are taught by Instructors. Both an Instructor and a Teaching Assistant are a specialization of an Approver. Students can form Teams within Classes. Initially, a Team is a Non-confirmed Team. An Approver approves a Non-confirmed Team if the team meets the requirements of an approved Team (e.g. number of students) then it can be

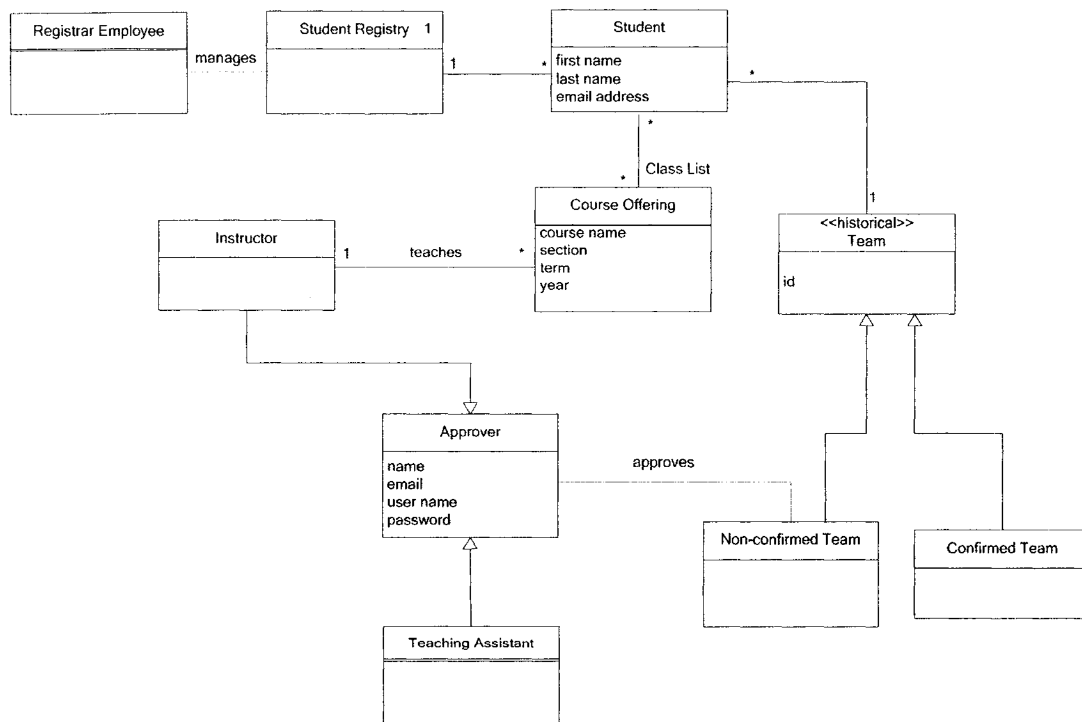


Figure 15. Domain model of course management system

updated to a Confirmed Team.

4.1.2 Actors

A list of Course Management System actors and their general responsibilities is presented in Table 2.

Actor	Responsibilities
Registrar Employee	• Views and/or updates the Student Registry.
Instructor	• Create and manage Class Lists. • Import Class List
Approver	• Manage Teams
Student	• Invite other Students to form a Team. • Confirm the Team membership. • Submit individual/team assignments.
Teaching Assistant	• Mark the assignments • Make comments on assignments.

Table 2. Actors and their responsibilities

4.1.3 Use case briefs

The expected functionality of the application is documented in the use case model. In order to provide a comprehensive overview, we first summarize each user-goal level use case in the form of use case briefs. The detailed use cases can be found in the next section and in the Appendix. Use case briefs are given in Table 3.

Use Case Name	Primary Actor	Summary
Add Students to Student Registry	Registrar Employee	Add Students to Student Registry.
Edit Student Information in Student Registry	Registrar Employee	Edit information of one Student in the Student Registry.
Review Student Information	Registrar Employee	Review information of Students in the Student Registry.
Import Student Registry	Registrar Employee	Import the Student Registry from a file.
Import Class List	Instructor	Import the entire Class List from a file.
Add Students to a Class List	Instructor	Add Student(s) from Student Registry to a Class List.
Remove Student from Class List	Instructor	Remove a Student from a Class List. Note that the Student information will be unchanged in the Student Registry.
Review class list	Instructor	Instructor reviews the information of Class Lists.
Accept Team Membership Invitation	Student	Student can accept an invitation for joining to a Non-Confirmed Team. If the invitation is accepted, the Student will be a member of the Non-Confirmed Team. Otherwise, the Student will be available for other Teams to be invited. A Student should not be a member of more than one Team at a time.
Invite Student(s) to form a Team	Student	A Student can invite other Students to form a Team (for the purpose of working together on a group assignment).
Review and Approve Non-confirmed Teams	Approver	Approver asks the System for the list of Non-confirmed Teams (for a given Course Offering). If Non-Confirmed Teams meet the requirements of a Confirmed Team, the Approver updates their status as a Confirmed Team.

Table 3. Use case briefs of course management system

4.1.4 Non-functional requirements

As the emphasis is on business transactions within use cases, we do not describe the non-functional requirements of the system in detail. The non-functional requirements of the Course Management System, such as performance issues, reliability, fault tolerance, etc. are compiled in a supplementary specification document that is not in the scope of the thesis.

4.2 Selected Use Cases in Detail

In this section, we highlight some important use cases of the Course Management System. These use cases have been chosen so as to showcase different types of concurrency management strategies.

Note that in the previous chapters, we numbered the use case main success scenario and extension steps. In this chapter, we use the same use case template given in Chapter 3 but replace step numbers with labels as the step identifier. Thus, each step has either an exclusive label, for cross reference, or no label at all, if there is no extension applicable for the step. We choose labeling because it is more convenient for users to remember the position of the extensions. A label starts with "{" and ends with "}" and is written in *blue italic text.*

The following three use cases are subfunction use cases of the "Manage Student Registry" user goal use case given in the Appendix A.

4.2.1 Use Case: Edit Student Information (optimistic)

The Registrar Employee manages the information of Students in the Student Registry. Figure 16 shows the "Edit Student Information" use case. The transaction starts when the primary actor identifies the Student whose information he/she wishes to edit. This Student forms the only transactional resource of the use case. The system then provides the information of the Student to be edited. The primary actor modifies the information and asks the system to finalize the update. The system locks the Student information, commits the changes and the transaction terminates successfully. Note that in the given

scenario, an optimistic concurrency strategy has been adopted as the locking occurs at the very end of the transaction, just before committing step.

Figure 16. Use case: edit student information in student registry

USE CASE: EDIT STUDENT INFORMATION IN STUDENT REGISTRY

PROPERTIES

Primary Actor: Registrar Employee
Goal: Registrar Employee **successfully edits the information of one** Student
Level: subfunction
Frame: One Student (the one whose information is to be updated).
Precondition: Primary Actor is authenticated and Student Registry is not empty.

MAIN SUCCESS SCENARIO

{Trigger} Primary Actor indicates that he/she wishes to edit the information of one Student in the Student Registry.
System acknowledges the Primary Actor's request.
TRANSACTION. TRANSACTIONAL RESOURCES: a Student whose information is to be updated. We refer to this Student as **<John>**.
{Select} Primary Actor identifies <John>.
{Edit} Primary Actor provides updated information for any of the attributes of <John> except his ID and System ensures that the provided information is valid.
{Lock} BEGIN RESOURCE LOCK: exclusive read/write.
System updates the Student Registry with the new Student information.
{Commit} COMMIT.
System notifies Primary Actor that changes were made successfully.
Use case ends successfully.

EXTENSIONS

{Trigger} to (excluding){Lock}a. Primary Actor indicates that he/she wishes to abort the transaction:

- TRANSACTION ABORT.
- Use case ends unsuccessfully. Transactional resources remain unchanged.

{Select}a. <John> does not exist in the Student Registry:

- System informs Primary Actor that <John> does not exist in the Student Registry.
- Use case resumes at *{Select}*.

{Edit}a. Information concerning <John> is invalid:

- System informs Primary Actor that information is invalid (and states why).
- Use case resumes at *{Edit}*.

{Edit} to (excluding){Lock}a. <John> is no longer in the Student Registry:

- System notifies Primary Actor that <John> is no longer in Student Registry.
- Use case resumes at *{Select}*.

{Edit} to (excluding){Lock}b. <John>'s information has been changed by another Registrar Employee:

- System notifies Primary Actor that information of <John> has been edited by another transaction and that current changes will be discarded.
- Use case resumes at *{Edit}*.

{Lock}a. Transactional resources are unavailable:

- System notifies Primary Actor that the transactional resources are currently unavailable.
- TRANSACTION ABORT.
- Use case ends unsuccessfully. Transactional resources remain unchanged.

{Commit}a. Failure persisting data:

- System notifies primary actor that the transaction could not be completed because the changes could not be persisted.
- TRANSACTION ABORT.
- Use case ends unsuccessfully. Transactional resources remain unchanged.

4.2.2 Use Case: Import Student Registry (pessimistic)

Sometimes a Registrar Employee needs to import Student Registry from an external file (e.g. after backup recovery). Student Registry can be either as a text or as an xml file. Registrar Employee specifies the source file and imports the Student Registry directly from the file. The use case covering this functionality is given in Figure 17. The Student Registry, all Class Lists, all Students and all Teams are locked at the beginning of the transaction. We employ pessimistic concurrency management for this example, because the chances of a conflict between this transaction and other transactions that are dealing with the same TRs are relatively high.

Figure 17. Use case: import student registry

USE CASE: IMPORT STUDENT REGISTRY

PROPERTIES

Primary Actor: Registrar Employee
Goal: Import the content of the Student Registry from an external source file.
Level: subfunction
Frame: Student Registry, All Students, All Class Lists, All Teams.

PRECONDITION

- Primary Actor is authenticated

MAIN SUCCESS SCENARIO

{Trigger} Primary Actor indicates that he/she wishes to import the content of the Student Registry from a particular file.

System acknowledges the Primary Actor's request.

TRANSACTION. TRANSACTIONAL RESOURCES: Student Registry, all Students, all Class Lists, all Teams.

{Lock} BEGIN RESOURCE LOCK: exclusive read/write.

System asks for confirmation to import the Student Registry from the file.

{Confirm} Primary Actor confirms.

{Commit} COMMIT.

System notifies Primary Actor that the Student Registry has been successfully imported (with its former contents, if any, replaced by the content of the student information in the contained file).

Use case ends successfully.

EXTENSIONS

{Trigger} to (excluding){Lock} a. Primary Actor indicates that he/she wishes to abort the transaction:

- TRANSACTION ABORT.
- Use case ends unsuccessfully. Transactional resources remain unchanged.

{Lock} to {Commit} a. Primary Actor indicates that he/she wishes to abort the transaction :

- System suspends the current operation and asks for confirmation.
- Primary Actor confirms.
- TRANSACTION ABORT.
- Use case ends unsuccessfully. Transactional resources remain unchanged.

{Confirm} a. Primary Actor cancels:

- TRANSACTION ABORT.
- Use case ends unsuccessfully. Transactional resources remain unchanged.

{Lock} a. Transactional resources are unavailable:

- System informs Primary Actor that the transactional resources are not currently available.
- TRANSACTION ABORT.
- Use case ends unsuccessfully. Transactional resources are unlocked and remain unchanged.

{Lock} to TRANSACTION a. Response timeout:

- TRANSACTION ABORT.
- Use case ends unsuccessfully. Transactional resources remain unchanged.

{Commit} a. Failure persisting data:

- System notifies primary actor that the transaction could not be completed because the changes could not be persisted
- TRANSACTION ABORT.
- Use case ends unsuccessfully. Transactional resources remain unchanged.

4.2.3 Use Case: Delete Students from Student Registry (hybrid)

Generally, student information is not removed from the Student Registry but occasionally an entry is created by mistake. Under such circumstances, we need to be able to delete the

student record. The use case described in this section covers this functionality. When the transaction starts, the primary actor selects one or more students that he/she wishes to delete. Only Students who are not a member of a Team or a Class List can be deleted. In order to validate this condition the System needs to lock the Student Registry, all Teams, all Class Lists and all Student records that have been selected for deletion. This transaction makes use of a hybrid concurrency management strategy: locking is done neither at the very beginning (pessimistic) nor at the very end (optimistic) of the transaction.

Figure 18. Use case: delete students from student registry

USE CASE: DELETE STUDENTS FROM STUDENT REGISTRY

PROPERTIES

Primary Actor: Registrar Employee
Goal: Remove one or more Students from Student Registry
Level: subfunction
Frame: Student Registry, All Students, All Class Lists, All Teams.

PRECONDITION

- Primary Actor is authenticated
- Student Registry is not empty

MAIN SUCCESS SCENARIO

{Trigger} Primary Actor indicates that he/she wishes to delete one or more Students from the Student Registry.
System acknowledges the Primary Actor's request.
TRANSACTION. TRANSACTIONAL RESOURCES: Student Registry, all Students who have been identified for deletion (we will call these Students as <Students to be deleted>), all Class Lists, all Teams.
{Select} Primary Actor identifies the <Students to be deleted> from the Student Registry.
{Lock} BEGIN RESOURCE LOCK: exclusive read/write.
{Member} System ensures that <Students to be deleted> are not members of any Class List and/or Team.
System asks for confirmation to delete <Students to be deleted> from the Student Registry.
{Confirm} Primary Actor confirms.
{Commit} COMMIT.
System notifies Primary Actor that the <Students to be deleted> have been successfully deleted.
Use case ends successfully.

EXTENSIONS

{Trigger} to (excluding){Lock}a. Primary Actor indicates that he/she wishes to abort the transaction:

- TRANSACTION ABORT.
- Use case ends unsuccessfully. Transactional resources remain unchanged.

{Lock} to {Commit}a. Primary Actor indicates that he/she wishes to abort the transaction:

- System suspends the current operation and asks for confirmation.
- Primary Actor confirms.
- TRANSACTION ABORT.
- Use case ends unsuccessfully. Transactional resources remain unchanged.

{Select} to (excluding){Lock}a. One or more of the selected Students do not exist:

- System notifies Primary Actor that one or more <Students to be deleted>s have already been deleted by another user.
- TRANSACTION ABORT.
- Use case ends unsuccessfully. Transactional resources remain unchanged.

{Select} to (excluding){Lock}b. The information of one or more Students have changed:

- System notifies Primary Actor that the information of one or more Students of <Students to be deleted> has changed.
- System provides the names of <Students to be deleted> whose information has changed and asks the Primary Actor if he/she still wishes to delete the Students.
- *{Delete_edited_student}* Primary Actor confirms.
- Use case resumes at *{Lock}*.

{Select} to {Delete_edited_student}a. Primary Actor cancels:

- TRANSACTION ABORT.
- Use case ends unsuccessfully. Transactional resources remain unchanged.

{Lock}a. Transactional resources are unavailable:

- System informs Primary Actor that the transactional resources are not currently available.
- TRANSACTION ABORT.
- Use case ends unsuccessfully. Transactional resources remain unchanged.

{Member}a. At least One Student of <Students to be deleted> is member of one/more Class Lists:

- System notifies Primary Actor that at least one Student of <Students to be deleted> is member of one/more Class Lists and/or Teams and identifies the names.
- TRANSACTION ABORT.
- Use case ends unsuccessfully. Transactional resources remain unchanged.

{Confirm}a. Primary Actor cancels deletion:

- TRANSACTION ABORT.
- Use case ends unsuccessfully. Transactional resources remain unchanged.

{Lock} to TRANSACTION a. Response timeout:

- TRANSACTION ABORT.
- Use case ends unsuccessfully. Transactional resources remain unchanged.

{Commit}a. Failure persisting data:

- System notifies primary actor that the transaction could not be completed because the changes could not be persisted.
- TRANSACTION ABORT.
- Use case ends unsuccessfully. Transactional resources remain unchanged.

4.3 Summary

In this chapter, we illustrated using realistic scenarios how our template could be usefully applied to capture the requirements of business transactions. All use cases described in this chapter have been chosen from Course Management System use case model which is given in Appendix A. In the next chapter, we share observations about how best to gather requirements of business transactions in order to be most effective at writing use cases.

5 Writing Effective Transactions within Use Cases: Heuristic Guidelines

This chapter presents a set of heuristic guidelines for capturing business transaction requirements within use cases. The presented guidelines are a reflection of our experiences gained from modeling the transactional and concurrency aspects of the Course Management System. In particular, we describe a guideline for each of the following tasks:

- Finding transactional resources.
- Finding transaction boundaries.
- Choosing an appropriate concurrency management strategy.

5.1 Finding Transactional Resources

Choosing the right set of transactional resources (TRs) for a given transaction can be difficult. The main challenge is in identifying all *relevant* resources while avoiding the inclusion of unnecessary resources since the latter is likely to lead to reduced system availability. As was mentioned in Chapter 3, TRs are domain concepts taken from a system's Domain Model. With a detailed Domain Model at hand, we have found the following heuristic helpful:

- Identify every domain object that is directly manipulated by the business transaction as a transactional resource.
- If 'A' has already been identified as a transactional resource then every domain entity that is directly related to 'A' by virtue of an aggregation or composition relationship

(in a domain model typically signified by <<is part of>>, <<consists of>>, <<has>> stereotypes) is a prime candidate for inclusion as a transactional resource as well.

In the use case: "Delete Students from Student Registry" (Figure 18 on page 48), the Registrar Employee wishes to delete Students from the Student Registry. The Students who are about to be deleted are the objects that are directly manipulated by the transaction. If we use Students as a starting point, the following additional TRs can be easily identified through consultation of the corresponding domain model (depicted in Figure 15 on page 41):

- **Team**: A Student **is** (potentially) **part of a** Team
- **Student Registry:** The Student Registry **consists of** Students
- **Class List**: Each Student **has a** Class List.

5.2 Finding Transaction Boundaries

Determining the boundaries of a transaction, that is, determining when a transaction begins and when it commits, is not always obvious especially when optimistic concurrency management is used. A business transaction does not always begin and end with a system step that modifies transactional resources. Instead, a business transaction may start with a user interaction. A general heuristic to determine which steps belong to the business transaction is as follows:

- Assume (regardless of the envisaged concurrency management strategy) a pessimistic concurrency management strategy and determine the steps for which the transactional resources need to be locked such that a concurrency conflict is impossible.

- Add all interaction steps between the Primary Actor and the system that lead to the identification transactional resources.

Set the transaction boundaries as follows: the transaction starts immediately prior the first identified step and commits immediately after the last one.

MAIN SUCCESS SCENARIO

{Trigger} Primary Actor indicates that he/she wishes to add one or more Students to the Class List (We will call these Students <Students to be added>.)
System acknowledges the Primary Actor's request.
{Select} Primary Actor Select a Class List.
{Enter_Info} Primary Actor provides information of <Students to be added> and the System ensures that the provided information is valid.
{Add} System adds <Students to be added> to the specific Class List.
System notifies Primary Actor that <Students to be added> have been successfully added.
Use case ends successfully.

Figure 19. Main success scenario of "add student(s) to class list" without transactional boundaries

As an example, we apply the given guideline to determine the transactional boundaries of the use case: "Add Student(s) to the Class List", of Section 4.1.3. Figure 19 depicts the main success scenario of the use case. It is assumed that the transactional boundaries are not yet determined. In order to find the transactional boundaries we assume a pessimistic concurrency scheme and determine the steps that need to be locked in order to rule out concurrency conflicts. In this case, the steps with the labels *{Enter_Info}* and *{Add}* are identified. Next, we determine the interaction steps that may not cause any concurrency conflicts, but which are needed for the successful commit of the business transaction. In this case, the step labeled *{Select}* is identified. According to the last instruction of the guideline, we can conclude that the transaction starts just before *{Select}* and ends after

{Add}. Figure 20 illustrates the main success scenario of the use case with identified transactional boundaries.

> *MAIN SUCCESS SCENARIO*
>
> *{Trigger}* Primary Actor indicates that he/she wishes to add one or more Students to the Class List (We will call these Students <Students to be added>.)
> System acknowledges the Primary Actor's request.
> TRANSACTION. TRANSACTIONAL RESOURCES: All Students, all Class Lists.
>   *{Select}* Primary Actor Select a Class List.
>   *{Enter_Info}* Primary Actor provides information of <Students to be added> and the System ensures that the provided information is valid.
>   BEGIN RESOURCE LOCK: exclusive read/write.
>   COMMIT.
> System notifies Primary Actor that <Students to be added> have been successfully added.
> Use case ends successfully.

Figure 20. Main success scenario of "add students(s) to class list" with transactional boundaries

5.3 Choosing a Concurrency Management Strategy

In this thesis, we have discussed three different concurrency management strategies, namely optimistic, pessimistic and hybrid concurrency management. The selection of one or the other strategy directly affects when, during the lifetime of a transaction, the transaction resources are to be locked. Within our use case template the locking of transactional resources is denoted by BEGIN RESOURCE LOCK. The beginning and the end of a transaction are denoted by steps TRANSACTION and COMMIT respectively.

The choice as to which concurrency management strategy is most appropriate depends on domain related aspects such as the likelihood of a conflict and its impact. In what

follows, we present a set of selection guidelines for each concurrency management strategy.

5.3.1 Optimistic Concurrency Management

An optimistic concurrency strategy is applicable when the likelihood of a concurrency conflict is relatively low and when the cost of recovering from a conflict is small [Fowler 2003]. Optimistic concurrency management allows different transactions to have access to data of the same transactional resources until one of them attempts to lock the transactional resources [Fowler 2003]. As a result, the liveliness of the system is increased, but, on the downside, more conflicts are possible. A use case specifies an optimistic concurrency strategy if the locking of transactional resources happens at the end of the transaction, just before the COMMIT step.

5.3.2 Pessimistic Concurrency Management

In contrast pessimistic concurrency management is appropriate when the likelihood of a conflict is high or if the cost recovering from of a conflict is unacceptable [Fowler 2003]. In a pessimistic concurrency strategy, all transactional resources are locked at the beginning of transaction. As a result, concurrency conflicts are prevented as "lost updates" or "inconsistent reads" are not possible. Technically, a pessimistic concurrency strategy is easier to implement as cumbersome conflict resolutions can be omitted. On the downside, a purely pessimistic concurrency management significantly affects the performance and availability of the system and as such, is rarely the preferred strategy multi-user systems. A use case specifies a pessimistic concurrency management scheme if the first TRANSACTION substep is a BEGIN RESOURCE LOCK.

5.3.3 Hybrid

We use a hybrid concurrency management strategy when neither an optimistic scheme nor pessimistic concurrency management scheme is applicable. In other words, the transactional resources are locked neither at the very beginning nor at the very end of transactions.

Often hybrid concurrency management is more practical than purely pessimistic or optimistic strategies. It offers a compromise between conflict prevention and conflict detection and thus gives us the possibility to balance usability vs. availability of application.

6 Conclusion and Future Work

The main motivation behind our research was the observation that business transaction modeling, and the associated concurrency management, is to great extent a domain activity and hence, should be tackled in the requirements phase. The state-of-art in writing use cases (commonly used to capture functional requirements), has not provided a proper means for an integrated specification of functional requirements and business transaction requirements. In this thesis, we have tackled this shortcoming by defining an extension for use cases that covers the modeling of business transactions and sagas. We have presented what we believe to be a well-integrated extension, which preserves the intuitive nature of use cases. The use case extension is relevant for the requirements specification of any (interactive) application where a common data store is concurrently accessed by multiple actors. As a prominent example of this, we highlighted enterprise applications for which modeling and processing business transactions is one of the most complex and most crucial aspects. In an enterprise application development, functional requirements and the strategy to manage concurrency are closely related. Our approach closely incorporates references to domain objects and support modeling of transactional recourses.

As a first step in the core contribution of this thesis, we set up a vocabulary which captures all relevant core concepts for modeling business transactions at the requirements level. Then, we present templates for the modeling of business transactions and sagas showing how their specification is expressed through the main success scenario as well as use case extensions. The usage of the templates is illustrated through detailed examples.

Finally, we introduced guidelines to assist software engineers in identifying (1) transaction boundaries, (2) transactional resources, and (3) the most appropriate concurrency strategy to be used in the writing of business transactions within the given use case template. Through the templates, we point out some common (though optional) use case extensions, to help software engineers remember important alternative behavior that arises, e.g., from the choice of particular concurrency management strategy. We have ensured that our proposed extension to use cases is both expressive and flexible enough by applying it to a comprehensive case study: a use case model of a Course Management System (given in Appendix A). As a secondary artifact of the research conducted in the context of this thesis we provide, in Appendix C, the XML use case meta-model that was used in the creation of the Course Management System.

As future work, we are aiming to further extend the presented meta-model such that it explicitly incorporates nested transactions in sagas. Another future avenue deals with the refinement of the current all-or-nothing philosophy of the resource lock step. Such a refinement can be performed in two dimensions: (1) allow delayed locking of transactional resources—for example, transactional resource "A" might be locked at the beginning of the transaction whereas transactional resource "B" is locked at the end of the transaction; (2) combine different types of locks—e.g., a transaction resource "A" may require an exclusive read/write lock whereas transaction resource "B" may only require an exclusive read lock. Finally, we wish to further validate our work by applying it to other case studies.

References

[Abbot 83]. Abbott, R. *Program Design by Informal English Descriptions.* Communications of the ACM, vol. 26(11), 1983.

[Alrifai *et al.* 2006]. Alrifai, M., P. Dolog and W. Nejdl, Transactions Concurrency Control in Web Service Environment, in Proceedings of *ECOWS'06*, pp. 109-118, 2006.

[Atkins & Coady 1992]. Atkins M.S. and *Coady* M.Y. *Adaptable Concurrency Control for Atomic Data Types. in ACM Transactions on Computer Systems - Vol* 10, Num 3, 1992.

[Banagala 2006]. Banagala, V. *Analysis of Transaction Problems Using the Problem Frames Approach.* in *International Conference on Software Engineering (ICSE).* Shanghai, China, 2006.

[Bennett et al. 2000]. Bennett, B., Hahm, B., Leff, A., Mikalsen, T., Rasmus, K., Rayfield, J., Rouvellou, I., *A Distributed Object Oriented Framework to Offer Transactional Support for Long Running Business Processes.* in *Middleware* 2000.

[Butler et al. 2005]. Butler, M., C. Ferreira, and M. Ng, *Precise Modelling of Compensating Business Transactions and its Application to BPEL.* Journal of Universal Computer Science, **11**(5), 2005.

[Cockburn 2001]. Cockburn, A., *Writing Effective Use Cases*. Agile software development series, Boston: Addison-Wesley, 2001.

[Correa & Werner 2004]. Correa, A.L. and C.M.L. Werner. *Precise specification and validation of transactional business software*. in *Requirements Engineering 2004*. Kyoto, Japan, 2004.

[Gomma 2005]. Gomaa, H. *Designing software product lines with UML: from use cases to pattern-based software architectures*. Addison-Wesley, 2005.

[Fowler 2003].Fowler, M., *Patterns of Enterprise Application Architecture*, Boston, MA: Addison-Wesley, 2003,

[Gallina & Mammar 2006]. Gallina, B., N. Guelfi, and A. Mammar, *Structuring Business Nested Processes Using UML 2.0 Activity Diagrams and Translating into XPDL*, Passau, Germany, 2006.

[Garcia-Molina & Salem 1987]. Garcia-Molina, H. and K. Salem. *Sagas*. in *1987 ACM SIGMOD international conference on Management of data*. San Francisco, California, United States: ACM Press, 1987.

[Jackson 2001]. Jackson, M., *Problem Frames*, London: Pearson, 2001.

[Jacobson 1992]. Jacobson, I., *Object-Oriented Software Engineering : A Use Case Driven Approach*. New York: ACM Press (Addison-Wesley Pub), 1992.

[Larman 2005]. Larman, C., *Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and the Unified Process*. 3rd ed., Upper Saddle River, NJ: Prentice Hall PTR, 2005.

[Lilly 1999]. Lilly, S. *Use Case Pitfalls: Top 10 Problems from Real Projects Using Use Cases*. in *TOOLS USA*. 1999.

[Merrick & Barrow 2005]. Merrick, P. and P. Barrow, *The Rationale for OO Associations in Use Case Modelling.* Journal of Object Technology,. **4**(9): p. 123-142, 2005.

[Overgaard & Palmkvist 2004]. Overgaard, G. and K. Palmkvist, Use Cases Patterns and Blueprints, Addison-Wesley, Indianapolis, 2004.

[Ramakrishnan & Gehrke 2002]. Ramakrishnan, R. and J. Gehrke, *Database Management Systems 3rd edition*. McGraw-Hill, 2002.

[Subrahmanyam & Richard 2000]. Subrahmanyam, A., K. A., and B. Richard, *Professional Java Server Programming J2EE Edition*, 2000.

[Toerner et al. 2006]. Toerner, F., et al. *An Empirical Quality Assessment of Automotive Use Cases*. in *14th IEEE International Requirements Engineering Conference 2006*. Minneapolis, Minnesota, USA, 2006.

[US Census 2006]. US Census Bureau, Quarterly Retail E-Commerce Sales. 3rd Quarter 2006. US Census Bureau News, November 17, 2006 (http://www.census.gov/mrts/www/data/html/06Q3.html).

[Whittle & Jayaraman 2006]. Whittle, J. and P.K. Jayaraman. *Generating Hierarchical State Machines from Use Case Charts*. in *14th IEEE International Requirements Engineering Conference 2006*. Minneapolis, Minnesota, USA, 2006.

Appendix A. Use Case Model: Course Management System

In this appendix, we present the use case model of the Course Management System that was introduced in Chapter 4. The use case model includes a domain model list, based on Figure 15, and three packages. All the use cases are written in XML in conformance to the UCM meta-model given in Appendix C and translated automatically into HTML using an XSL transformation script. In the HTML version, all definitions are hyperlinked, facilitating navigation. As a final note we point out that our transaction template has evolved over time and that the use case model does not reflect the most recent syntactic conventions, though it is accurate structurally.

DOMAIN MODEL LIST

This section lists the key conceptual classes of the *Domain Model* that are used in this Use Case Model.

- **Student Registry:** The central registry that contains the information of all students in the institution.
- **Student:** A student of the university.
- **Class List:** List of Classes
- **Team:** A group of 4-6 Students
- **Non-confirmed Team:** a Team which is not confirmed by approver
- **Confirmed Team:** a Team which is confirmed by approver

PACKAGE: MANAGE STUDENT REGISTRY

ACTOR LIST

Registrar Employee: Registrar employee who has the authority to make changes to the student registry.

USE CASE: MANAGE STUDENT REGISTRY

PROPERTIES

- **Primary Actor:** Registrar Employee
- **Goal:** Registrar Employee views and/or updates the Student Registry.
- **Level:** user-goal
- **Frame:**
  - Student Registry
  - All Students
  - All Class List Lists
  - All Teams
- **Precondition:**
  - Primary Actor is authenticated.

MAIN SUCCESS SCENARIO

- Primary Actor indicates that he/she wishes to view or update the Student Registry.
- System acknowledges the Primary Actor's request.
- Primary Actor repeatedly perform(s) the following step(s) until he/she is done viewing/updating the Student Registry
  - Primary Actor *chooses one of the following:*
    - Add Students to Student Registry.
    - Delete Students from Student Registry.
    - Edit Student Information in Student Registry.
    - Review Student Information.

USE CASE: ADD STUDENTS TO STUDENT REGISTRY

PROPERTIES

- **Primary Actor:** Registrar Employee
- **Goal:** Registrar Employee successfully adds Students to Student Registry .
- **Level:** subfunction
- **Frame:**
  - Student Registry
  - All Students
- **Precondition:**
  - Primary Actor is authenticated

Main Success Scenario

- *{Trigger}* Primary Actor indicates that he/she wishes to add one or more Students to the Student Registry (We will call these Students **<Students to be added>**.)
- System acknowledges the Primary Actor's request.
- *{Begin Transaction}*Transactional resources:Student Registry, all Students.
    - *{Enter_Validate_Info}* Primary Actor provides information of <Students to be added> and the System ensures that the provided information is valid.
    - *{Lock}* exclusive read/write.
    - *{Commit}* System adds <Students to be added> to the Student Registry.
- System notifies Primary Actor that <Students to be added> have been successfully added.
- Use case ends successfully.

Extensions

- **{Trigger} to (excluding){Lock}a.** Primary Actor indicates that he/she wishes to abort the transaction:
    - Transaction Abort.
    - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Enter_Validate_Info}a.** Information of <Students to be added> is invalid:
    - System informs the Primary Actor that information of <Students to be added> is invalid (and states why).
    - Use case resumes at {Enter_Validate_Info}.
- **{Enter_Validate_Info} to (excluding){Lock}a.** Among the <Students to be added> , one or more Student IDs is/are already in use in the Student Registry:
    - System informs Primary Actor that <Students to be added> with the provided Student ID already exists in the Student Registry.
    - Use case resumes at {Enter_Validate_Info}.
- **{Lock}a.** Transactional resources are unavailable:
    - System notifies Primary Actor that the transactional resources are currently unavailable.
    - Transaction Abort.
    - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Commit}a.** Failure persisting data:
    - Transaction Abort.
    - Use case ends unsuccessfully. Transactional resources remain unchanged.

Use Case: Delete Students from Student Registry

Properties

- **Primary Actor:** Registrar Employee
- **Goal:** Registrar Employee successfully deletes Students from Student Registry

- **Level:** subfunction
- **Minimal Guarantee:**
- **Frame:**
  - Student Registry
  - All Students
  - All Class Lists
  - All Teams
- **Precondition:**
  - Primary Actor is authenticated
  - Student Registry is not empty

MAIN SUCCESS SCENARIO

- *{Trigger}* Primary Actor indicates that he/she wishes to delete one or more Students from the Student Registry.
- System acknowledges the Primary Actor's request.
- *{Begin Transaction}*Transactional resources:Student Registry, all Students who have been identified for deletion (we will call these Students as **<Students to be deleted>**), all Class Lists, all Teams.
  - *{Selection}* Primary Actor identifies that he/she wishes to delete <Students to be deleted> from the Student Registry.
  - *{Lock}* exclusive read/write.
  - *{Member}* System ensures that <Students to be deleted> are not members of any Class List and/or Team.
  - System asks for confirmation to delete <Students to be deleted> from the Student Registry.
  - *{Confirm}* Primary Actor confirms.
  - *{Commit}* System deletes <Students to be deleted> from Student Registry.
- System notifies Primary Actor that the <Students to be deleted> have been successfully deleted.
- Use case ends successfully.

EXTENSIONS

- **{Trigger} to (excluding){Lock}a.** Primary Actor aborts the transaction:
  - Transaction Abort.
  - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Lock} to {Commit}a.** Primary Actor aborts the transaction:
  - System suspends the current operation and asks for confirmation.
  - *{Confirm_Delete}* Primary Actor confirms.
  - Transaction Abort.
  - Use case ends unsuccessfully. transactional resources remain unchanged.
- **{Selection} to (excluding){Lock}a.** One or more of the selected Students do not exist:

- System notifies Primary Actor that one or more <Students to be deleted>s have already been deleted by another user.
  - Transaction Abort.
  - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Selection} to (excluding){Lock}b.** The information of one or more Students have changed:
  - System notifies Primary Actor that the information of one or more Students of <Students to be deleted> has changed.
  - System provides the names of <Students to be deleted> whose information has changed and asks the Primary Actor if he/she still wishes to delete the Students.
  - *{Delete_edited_student}* Primary Actor confirms.
  - Use case resumes at {Commit}.
- **{Selection} to {Delete_edited_student}a.** Primary Actor cancels:
  - Transaction Abort.
  - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Lock}a.** Transactional resources are unavailable:
  - System informs Primary Actor that the transactional resources are not currently available.
  - Transaction Abort.
  - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Member}a.** At least One student of <Students to be deleted> is member of one/more Class Lists:
  - System notifies Primary Actor that at least one student of <Students to be deleted> is member of one/more Class Lists and/or Teams and identifies the names.
  - Transaction Abort.
  - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Confirm}a.** Primary Actor cancels deletion:
  - Transaction Abort.
  - Use case ends unsuccessfully. Transactional resources and <Students to be deleted> remain unchanged.
- **{Lock} to (excluding){Begin Transaction}a.** Response time-out:
  - Transaction Abort.
  - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Commit}a.** Failure persisting data:
  - Transaction Abort.
  - Use case ends unsuccessfully. Transactional resources remain unchanged.

USE CASE: EDIT STUDENT INFORMATION IN STUDENT REGISTRY

PROPERTIES

- **Primary Actor:** Registrar Employee
- **Goal:** Registrar Employee successfully edits the information of one Student
- **Level:** subfunction
- **Minimal Guarantee:**
- **Frame:**
    - One Student (the one whose information is to be updated)
- **Precondition:**
    - Primary Actor is authenticated
    - Student Registry is not empty.

MAIN SUCCESS SCENARIO

- *{Trigger}* Primary Actor indicates that he/she wishes to edit the information of one Student in the Student Registry.
- System acknowledges the Primary Actor's request.
- *{Begin Transaction}*Transactional resources: a Student whose information is to be updated. We refer to this Student as **<John>** .
    - *{Select_Student}* Primary Actor identifies <John>.
    - *{Edit_Validate_ Info}* Primary Actor provides updated information for any of the attributes of <John> except his ID, and System ensures that the provided information is valid:
    - *{Lock}* exclusive read/write.
    - *{Commit}* System updates the Student Registry with the new Student information.
- System notifies Primary Actor that changes were made successfully.
- Use case ends successfully.

EXTENSIONS

- **{Trigger} to (excluding){Lock}a.** Primary Actor aborts the transaction:
    - Transaction Abort.
    - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Select_Student}a.** <John> does not exist in the Student Registry:
    - System informs Primary Actor that <John> does not exists in the Student Registry.
    - Use case resumes at {Select_Student}.
- **{Edit_Validate_ Info}a.** Information of <John> is invalid:
    - System informs Primary Actor that information is invalid (and states why).
    - Use case resumes at {Edit_Validate_ Info}.
- **{Edit_Validate_ Info} to (excluding){Lock}a.** <John> is no longer in the Student Registry:

- System notifies Primary Actor that <John> is no longer in Student Registry.
- Transaction Abort.
- Use case ends unsuccessfully. Transactional resources remain unchanged.

- **{Edit_Validate_ Info} to (excluding){Lock}b.** <John>'s information has been changed by another Registrar Employee:
  - System notifies Primary Actor that information of <John> has been edited by another transaction and that current changes will be discarded.
  - Transaction Abort.
  - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Lock}a.** Transactional resources are unavailable:
  - System notifies Primary Actor that the transactional resources are currently unavailable.
  - Transaction Abort.
  - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Commit}a.** Failure persisting data:
  - System notifies primary actor that the transaction could not be completed because the changes could not be persisted
  - Transaction Abort.
  - Use case ends unsuccessfully. Transactional resources remain unchanged.

USE CASE: REVIEW STUDENT INFORMATION

PROPERTIES

- **Primary Actor:** Registrar Employee
- **Goal:** Review the Students in Student Registry .
- **Level:** subfunction
- **Precondition:**
  - Primary Actor is authenticated.

MAIN SUCCESS SCENARIO

- *{Trigger}* Primary Actor indicates that he/she wishes to review the Student Registry and System acknowledges the Primary Actor's request.
- *{Begin Transaction}*Transactional resources:Student Registry, all Student s who have been identified for Reviewing. We will call these students **<Students to be reviewed>**.
  - *{Selection}* Primary Actor identifies the <Students to be reviewed> by providing the selection criteria.
  - *{Lock}* read.
  - *{Create View}* System provides information of <Students to be reviewed> from the Student Registry.

    - *{Commit}*
- *{Navigation}* Primary Actor reviews and paging through the <Students to be reviewed>.
- Use case ends successfully.

EXTENSIONS

- **{Trigger} to (excluding){Lock}a.** Primary Actor indicates that he/she wishes to abort the transaction:
    - Transaction Abort.
    - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Lock}a.** Transactional resources are unavailable:
    - System notifies Primary Actor that the transactional resources are currently unavailable.
    - Transaction Abort.
    - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Create View}a.** System fails to retrieve persistent data:
    - System notifies Primary Actor of the failure.
    - Transaction Abort.
    - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Navigation}a.** Primary Actor indicates that he/she wishes to change the selection criteria for <Students to be reviewed> :
    - Use case resumes at {Begin Transaction}.
- **{Navigation}b.** Primary Actor or the System refreshes/reloads the information of <Students to be reviewed> (due to freshness time outs, data chunk reload, etc).
    - *{System_Transaction}*Transactional resources: Student Registry , <Students to be reviewed> .
        - *{System_Begin Resource Lock}* System locks transactional resources.
        - *{System_Create View}* System provides information of <Students to be reviewed> from the Student Registry based on the given selection criteria.
        - Use case resumes at {Navigation}.

PACKAGE: MANAGE CLASS LIST

ACTOR LIST

- **Instructor**: Instructor who manages the Class List.

USE CASE: IMPORT CLASS LIST

PROPERTIES

- **Primary Actor:** Instructor
- **Goal:** Primary actor successfully adds Class List .
- **Level:** user-goal
- **Frame:**
  - Student Registry
  - All Students
  - All Class Lists
  - All Teams
- **Precondition:**
  - Primary actor is authenticated
  - Student Registry is not empty

MAIN SUCCESS SCENARIO

- *{Trigger}* Primary Actor Primary actor indicates that he/she wishes to create a Pre-Registered Class List.
- System acknowledges the Primary Actor's request.
- *{Begin Transaction}*Transactional resources:all Student, , all Class Lists , and all Teams. .
  - *{Enter_Validate_Info}* Primary Actor provides information of Class List and the System ensures that the provided information is valid by valid by (1) checking that all mandatory information has been provided, (2) that the information has the right format, (3) and the information is within the allowed boundaries.
  - *{Lock}* exclusive read/write.
  - *{Commit}* System imports Class List
- System notifies Primary Actor that Class List has been successfully imported.
- Use case ends successfully.

EXTENSIONS

- **{Trigger} to (excluding){Lock}a.** Primary Actor indicates that he/she wishes to abort the transaction:
  - Transaction Abort.
  - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Enter_Validate_Info}a.** Information of Class List is invalid:
  - System notifies Primary Actor that information of Class List is invalid (and states why).
  - Use case resumes at {Enter_Validate_Info}.
- **{Lock}a.** Transactional resources are unavailable:
  - System notifies Primary Actor that the transactional resources are currently unavailable.

    - Transaction Abort.
    - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Commit}a.** Failure persisting data:
    - Transaction Abort.
    - Use case ends unsuccessfully. Transactional resources remain unchanged.

Use Case: Add Student(s) to a Class List

Properties

- **Primary Actor:** Instructor
- **Goal:** Instructor successfully adds Students to Class List .
- **Level:** user-goal
- **Frame:**
    - Student Registry
    - All Students
    - and All Teams.
- **Precondition:**
    - Primary Actor is authenticated
    - Class List exists, already

Main Success Scenario

- *{Trigger}* Primary Actor indicates that he/she wishes to add one or more Students to the Class List (We will call these Students **<Students to be added>**.)
- System acknowledges the Primary Actor's request.
- *{Begin Tra*nsaction}Transactional resources:All Students, all Class Lists.
    - *{Select_Class_List}* Primary Actor Select a Class List.
    - *{Enter_Validate_Info}* Primary Actor provides information of <Students to be added> and the System ensures that the provided information is valid.
    - *{Lock}* exclusive read/write.
    - *{Commit}* System adds <Students to be added> to the specific Class List.
- System notifies Primary Actor that <Students to be added> have been successfully added.
- Use case ends successfully.

Extensions

- **{Trigger} to (excluding){Lock}a.** Primary Actor indicates that he/she wishes to abort the transaction:
    - Transaction Abort.
    - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Enter_Validate_Info}a.** Information of <Students to be added> is invalid:

    - System informs the Primary Actor that information of <Students to be added> is invalid (and states why).
    - Use case resumes at {Enter_Validate_Info}.
- **{Enter_Validate_Info} to (excluding){Lock}a.** Among the <Students to be added> , one or more Student IDs is/are already in Class List:
    - System informs Primary Actor that <Students to be added> with the provided Student ID already exists in the Class List.
    - Use case resumes at {Enter_Validate_Info}.
- **{Lock}a.** Transactional resources are unavailable:
    - System notifies Primary Actor that the transactional resources are currently unavailable.
    - Transaction Abort.
    - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Commit}a.** Failure persisting data:
    - Transaction Abort.
    - Use case ends unsuccessfully. Transactional resources remain unchanged.

USE CASE: DELETE STUDENTS FROM CLASS LIST

PROPERTIES

- **Primary Actor:** Instructor
- **Goal:** Instructor successfully deletesStudent s from Class List
- **Level:** user-goal
- **Minimal Guarantee:**
- **Frame:**
    - All Students
    - All Class Lists
    - All Teams
- **Precondition:**
    - Primary Actor is authenticated
    - Student Registry is not empty

MAIN SUCCESS SCENARIO

- *{Trigger}* Primary Actor indicates that he/she wishes to delete one or more Students from the Class List. will call these Students (**<Students to be deleted>**).
- System acknowledges the Primary Actor's request.
- *{Begin Transaction}*Transactional resources:all Students who have been identified for deletion (we will call these Students as **<Students to be deleted>**), all Class Lists, all Teams.
    - *{Select_Class_List}* Primary Actor Select a Class List.
    - *{Selection}* Primary Actor identifies that he/she wishes to delete <Students to be deleted> from the Class List.
    - *{Lock}* exclusive read/write.

  - *{Member}* System ensures that <Students to be deleted> are not members of any Team.
  - System asks for confirmation to delete <Students to be deleted> from the Class List.
  - *{Confirm}* Primary Actor confirms.
  - *{Commit}* System deletes <Students to be deleted> from Class List.
- System notifies Primary Actor that the <Students to be deleted> have been successfully deleted.
- Use case ends successfully.

EXTENSIONS

- **{Trigger} to (excluding){Lock}a.** Primary Actor aborts the transaction:
  - Transaction Abort.
  - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Lock} to (excluding){Commit}a.** Primary Actor aborts the transaction:
  - System suspends the current operation and asks for confirmation.
  - *{Confirm_Delete}* Primary Actor confirms.
  - Transaction Abort.
  - Use case ends unsuccessfully. transactional resources remain unchanged.
- **{Selection} to (excluding){Lock}a.** One or more of the selected Students do not exist:
  - System notifies Primary Actor that one or more <Students to be deleted>s have already been deleted by another user.
  - Transaction Abort.
  - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Selection} to (excluding){Lock}b.** The information of one or more Students have changed:
  - System notifies Primary Actor that the information of one or more Students of <Students to be deleted> has changed.
  - System provides the names of <Students to be deleted> whose information has changed and asks the Primary Actor if he/she still wishes to delete the Students.
  - *{Delete_edited_student}* Primary Actor confirms.
  - Use case resumes at {Commit}.
- **{Selection} to {Delete_edited_student}a.** Primary Actor cancels:
  - Transaction Abort.
  - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Lock}a.** Transactional resources are unavailable:
  - System informs Primary Actor that the transactional resources are not currently available.
  - Transaction Abort.

    - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Member}a.** At least One student of <Students to be deleted> is member of one/more Class Lists:
    - System notifies Primary Actor that at least one student of <Students to be deleted> is member of one/more Class Lists and/or Teams and identifies the names.
    - Transaction Abort.
    - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Confirm}a.** Primary Actor cancels deletion:
    - Transaction Abort.
    - Use case ends unsuccessfully. Transactional resources and <Students to be deleted> remain unchanged.
- **{Lock} to {Begin Transaction}a.** Response time-out:
    - Transaction Abort.
    - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Commit}a.** Failure persisting data:
    - Transaction Abort.
    - Use case ends unsuccessfully. Transactional resources remain unchanged.

Use Case: Review Class List

Properties

- **Primary Actor:** Instructor
- **Goal:** Reviews a Class List .
- **Level:** subfunction
- **Precondition:**
    - Primary Actor is authenticated.

Main Success Scenario

- *{Trigger}* Primary Actor indicates that he/she wishes to review the Class List and System acknowledges the Primary Actor's request.
- *{Begin Transaction}*Transactional resources:All Class List s which have been identified for Reviewing. We will call these **<Class List to be reviewed>**.
    - *{Selection}* Primary Actor identifies the <Class List to be reviewed> by providing the selection criteria.
    - *{Lock}* exclusive read/write.
    - *{Commit}* System provides information of <Class List to be reviewed> from the Class List.
- *{Navigation}* Primary Actor reviews and paging through the <Class List to be reviewed>.
- Use case ends successfully.

EXTENSIONS

- **{Trigger} to {Lock}a.** Primary Actor indicates that he/she wishes to abort the transaction:
  - Transaction Abort.
  - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Lock}a.** Transactional resources are unavailable:
  - System notifies Primary Actor that the transactional resources are currently unavailable.
  - Transaction Abort.
  - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Commit}a.** Failure persisting data:
  - System notifies Primary Actor of the failure.
  - Transaction Abort.
  - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Navigation}a.** Primary Actor indicates that he/she wishes to change the selection criteria for <Class List to be reviewed> :
  - Use case resumes at {Begin Transaction}.
- **{Navigation}b.** Primary Actor or the System refreshes/reloads the information of <Class List to be reviewed> (due to freshness time outs, data chunk reload, etc).
  - *{System_Transaction}*Transactional resources: Class List .
    - *{System_Begin Resource Lock}* System locks transactional resources.
    - *{System_Create View}* System provides information of <Class List to be reviewed> from the Class List based on the given selection criteria.
    - Use case resumes at {Navigation}.

PACKAGE: MANAGE TEAMS

ACTOR LIST

- **Student**: Student who manages his/her Team membership.
- **Approver**: Approver approves a Non-confirmed Team to make it as a Confirmed Team.

USE CASE: ACCEPT TEAM MEMBERSHIP INVITATION

PROPERTIES

- **Primary Actor:** Student
- **Goal:** Primary Actor successfully Accept an invitation and becomes member of a Team.
- **Level:** user-goal

- **Frame:**
  - All Students
  - All Teams
  - All Class Lists
- **Precondition:**
  - Primary Actor is authenticated
  - Class List is not empty

MAIN SUCCESS SCENARIO

- *{Trigger}* Primary Actor indicates that he/she wishes to accept an invitation to become a member of a Team.
- System acknowledges the Primary Actor's request.
- *{Begin Transaction}*Transactional resources:the Student who attempts to accept an invitation(we call this Student as **<John>**), all Teams.
  - *{Select_Invitation}* Primary Actor identifies an invitation and asks System to join the Team.
  - *{Lock}* exclusive read/write.
  - *{Check_Availability}* System ensures that the identified Team is not a Confirmed Team.
  - *{Member}* System ensures that the <John> is not member of any other Teams.
  - *{Commit}* System adds <John> as a member of the Team.
- System notifies Primary Actor that the invitation is accepted and <John> becomes a member of the Team.
- Use case ends successfully.

EXTENSIONS

- **{Trigger} to (excluding){Lock}a.** Primary Actor indicates that he/she wishes to abort the transaction:
  - Transaction Abort.
  - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Check_Availability}a.** System indicates that the Team is a Confirmed Team and the number of student in the Team is not full:
  - System informs Primary Actor that he/she cannot be a member of the Team and states why.
  - Transaction Abort.
  - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Member}a.** System identifies that the Primary Actor is already a member of other Team:
  - System informs Primary Actor that <John> is already a member of other Teams and provides the information of his Team.
  - Transaction Abort.

- Use case ends unsuccessfully. Transactional resources remain unchanged.

- **{Lock}a.** Transactional resources are unavailable:
  - System notifies Primary Actor that the transactional resources are currently unavailable.
  - Transaction Abort.
  - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Commit}a.** Failure persisting data:
  - Transaction Abort.
  - Use case ends unsuccessfully. Transactional resources remain unchanged.

USE CASE: INVITE STUDENT(S) TO FORM A TEAM

PROPERTIES

- **Primary Actor:** Student
- **Goal:** Primary Actor successfully sends an invitation to Students to establish a Team.
- **Level:** user-goal
- **Frame:**
  - Student Registry
  - All Class Lists
  - All Teams
- **Precondition:**
  - Primary Actor is authenticated
  - Class List is not empty

MAIN SUCCESS SCENARIO

- *{Trigger}* Primary Actor indicates that he/she wishes to send an invitation to Students of a Class List to establish a Team.
- System acknowledges the Primary Actor's request.
- *{Begin Transaction}*Transactional resources:Student Registry, all Class Lists, all Teams.
  - *{Select_Class List}* Primary Actor identifies a Class List.
  - *{Lock}* exclusive read/write.
  - *{Member}* System provides information of Student of the selected Class List who are neither in a Non-confirmed Team nor in a Confirmed Team.
  - *{Commit}*
- Primary Actor selects the Students from the list and asks System to send them an invitation by email.
- System sends the invitation by email.
- Use case ends successfully.

Extensions

- **{Trigger} to (excluding){Lock}a.** Primary Actor indicates that he/she wishes to abort the transaction:
    - Transaction Abort.
    - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Member}a.** System indicates that the Team is a Confirmed Team:
    - System System informs Primary Actor that he/she cannot be a member of the Team and states why.
    - Transaction Abort.
    - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Commit}a.** Failure persisting data:
    - Transaction Abort.
    - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Lock}a.** Transactional resources are unavailable:
    - System notifies Primary Actor that the transactional resources are currently unavailable.
    - Transaction Abort.
    - Use case ends unsuccessfully. Transactional resources remain unchanged.

Use Case: Review and Approve Non-confirmed Teams

Properties

- **Primary Actor:** Approver
- **Goal:** Primary Actor successfully approves a Non-confirmed Team to make it as a Confirmed Team.
- **Level:** user-goal
- **Frame:**
    - Student Registry
    - All Class Lists
    - All Teams
- **Precondition:**
    - Primary Actor is authenticated

Main Success Scenario

- *{Trigger}* Primary Actor indicates that he/she wishes to approve Non-confirmed Teams.
- System acknowledges the Primary Actor's request.
- *{Begin Transaction}*Transactional resources:all Class Lists, all Teams.
    - *{Lock}* exclusive read/write.

    - *{Select_Update}* Primary Actor updates the Non-confirmed Teams as Confirmed Teams.
    - *{Commit}* System updates the information of selected Non-confirmed Teams to Confirmed Teams.
- System notifies Primary Actor that the information of Non-confirmed Teams are updated as Confirmed Teams.
- Use case ends successfully.

EXTENSIONS

- **{Trigger} to (excluding){Lock}a.** Primary Actor indicates that he/she wishes to abort the transaction:
    - Transaction Abort.
    - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Lock}a.** Transactional resources are unavailable:
    - System notifies Primary Actor that the transactional resources are currently unavailable.
    - Transaction Abort.
    - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Commit}a.** Failure persisting data:
    - Transaction Abort.
    - Use case ends unsuccessfully. Transactional resources remain unchanged.
- **{Lock} to {Begin Transaction}a.** Respone time-out:
    - Transaction Abort.
    - Use case ends unsuccessfully. transactional resources remain unchanged.

Appendix B. Domain Model

This appendix offers some discussion surrounding the concept of "domain model". Fowler defines it as follows: "*A Domain Model is an object model of the domain that incorporates both behavior and data .... A Domain Model creates a web of interconnected objects, where each object represents some meaningful individual, whether as large as a corporation or as small as a single line on an order form*" [Fowler 2003]. A domain model consists of two different kinds of object: (1) objects that mimic data in the business, and (2) objects represent the rules of the business uses [Fowler 2003]. Although an object oriented domain model looks like a database model, there is a significant difference between them. Objects in database model are data. But we combine data and process as a multi-valued attributes and uses inheritance in the domain model. Fowler defines two different types of domain model [Fowler 2003]:

- Simple domain model.
- Rich domain model.

Simple domain model: looks like the database model. Typically one domain object exists for each database table. Figure 21 depicts an example of a simple domain model.

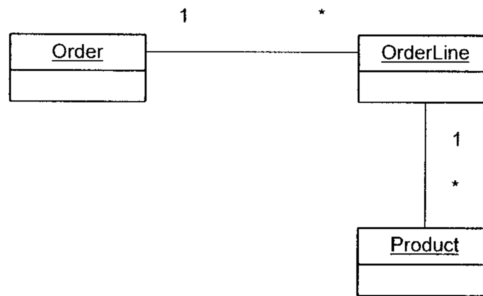


Figure 21. Example of a simple domain model

Rich domain Model: it is more complex and different from database model. It consists of inheritance, design patterns, and strategies [Fowler 2003]. Figure 22 illustrates a sample of a rich domain model.

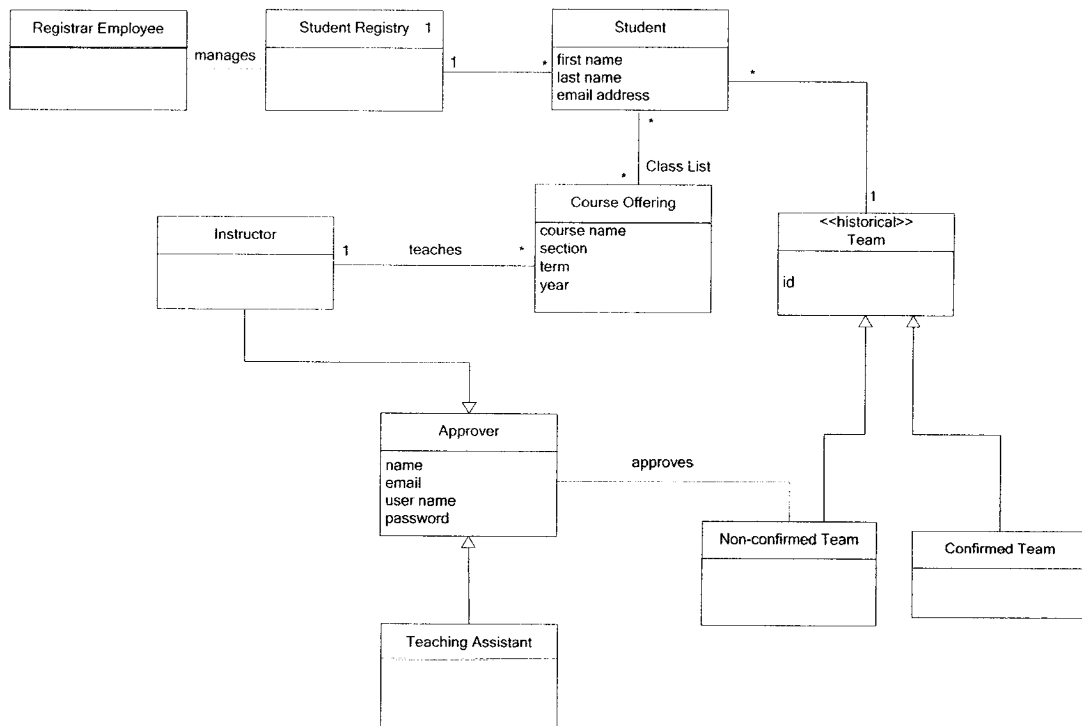


Figure 22. Example of a rich domain model

Appendix C. An XML UCM Meta-model for use cases with Eclipse

We developed a use case meta-model, based on the template given in Chapter 3, in the form of an XML DTD. Through this DTD, we define a use case model to consist of a collection of packages where a package includes the details of its actors and use cases. An XSL (eXtensible Stylesheet Language) transform can be applied to DTD conformant XML use case models to obtain an HTML format as output. The complete list of DTD elements are as following (Altova XMLSpy® 2007 was used to generate the DTD documentation format given next):

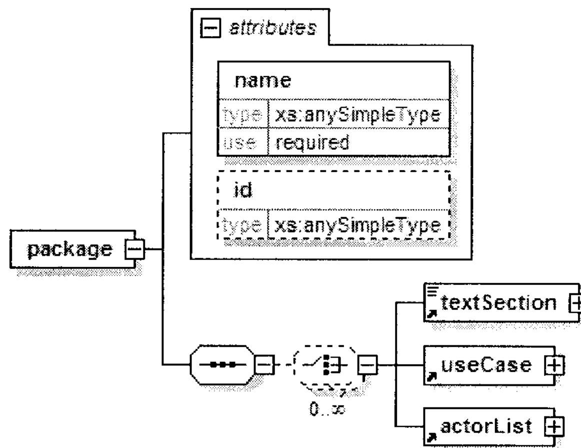
Actor	frame	stepAnyOrder
actorIsA	glossary	stepChoice
actorList	glossaryEntry	stepEnd
actorRef	goal	stepGoto
condition	level	stepRef
description	mainSuccessScenario	stepRepeat
domainModelEntity	minimalGuarantee	stepSeq
domainModelList	nameRef	stepTransaction
domainModelRef	package	systemRef
event	precondition	textSection
extension	primaryActor	transactionalResources
extensionPair	properties	ucm
extensionPoint	repeatCond	useCase
extensionRegion	resourceName	useCaseRef
extensions	step	varDef
		varRef

Table 4. List of DTD elements

In what follows, we present the detail of some important elements:

Package

diagram



properties	Content	complex		
children	**textSection useCase actorList**			
used by	element	**ucm**		
attributes	Name	Type	Use	Default
	name	**xs:anySimpleType**	required	
	id	**xs:anySimpleType**		

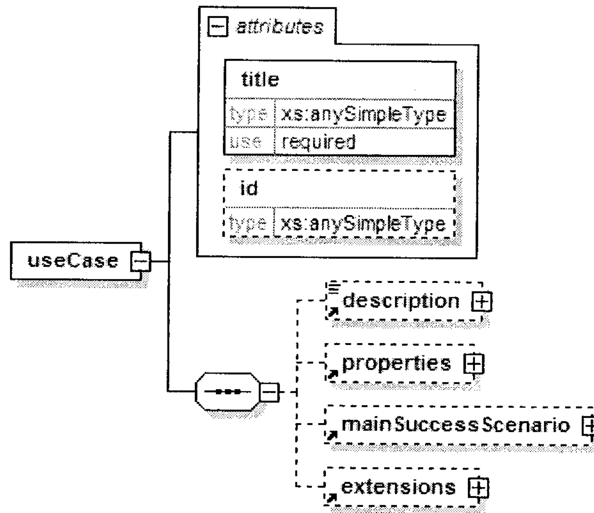
source

```
<xs:element name="package">
 <xs:complexType>
  <xs:sequence>
   <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element ref="textSection"/>
    <xs:element ref="useCase"/>
    <xs:element ref="actorList"/>
   </xs:choice>
  </xs:sequence>
  <xs:attribute name="name" type="xs:anySimpleType"
    use="required"/>
  <xs:attribute name="id" type="xs:anySimpleType"/>
 </xs:complexType>
</xs:element>
```

Figure 23. Structure of the package

Use Case

diagram



properties: Content complex

children: **description** **properties** **mainSuccessScenario** **extensions**

used by: element **package**

attributes

Name	Type	Use	Default
title	**xs:anySimpleType**	required	
id	**xs:anySimpleType**		

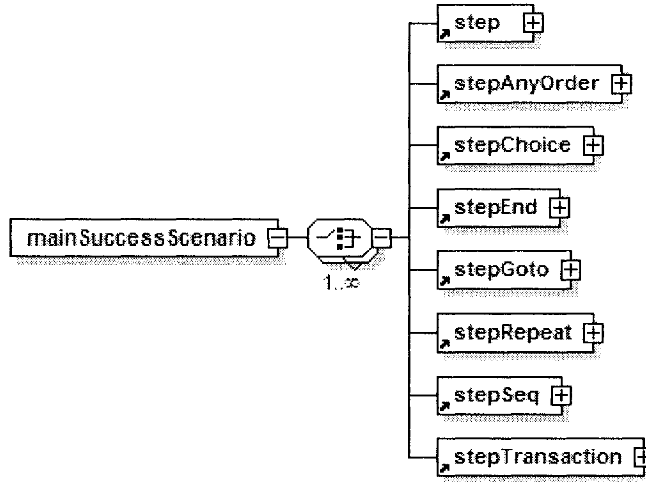
source

```
<xs:element name="useCase">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="description" minOccurs="0"/>
      <xs:element ref="properties" minOccurs="0"/>
      <xs:element ref="mainSuccessScenario" minOccurs="0"/>
      <xs:element ref="extensions" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="title" type="xs:anySimpleType"
      use="required"/>
    <xs:attribute name="id" type="xs:anySimpleType"/>
  </xs:complexType>
</xs:element>
```

Figure 24. Structure of the use case

Main Success Scenario

diagram



properties content complex

children **step stepAnyOrder stepChoice stepEnd stepGoto stepRepeat stepSeq stepTransaction**

used by element **useCase**

source

```
<xs:element name="mainSuccessScenario">
 <xs:complexType>
  <xs:choice maxOccurs="unbounded">
   <xs:element ref="step"/>
   <xs:element ref="stepAnyOrder"/>
   <xs:element ref="stepChoice"/>
   <xs:element ref="stepEnd"/>
   <xs:element ref="stepGoto"/>
   <xs:element ref="stepRepeat"/>
   <xs:element ref="stepSeq"/>
   <xs:element ref="stepTransaction"/>
  </xs:choice>
 </xs:complexType>
</xs:element>
```

Figure 25. Structure of the main success scenario

Extensions

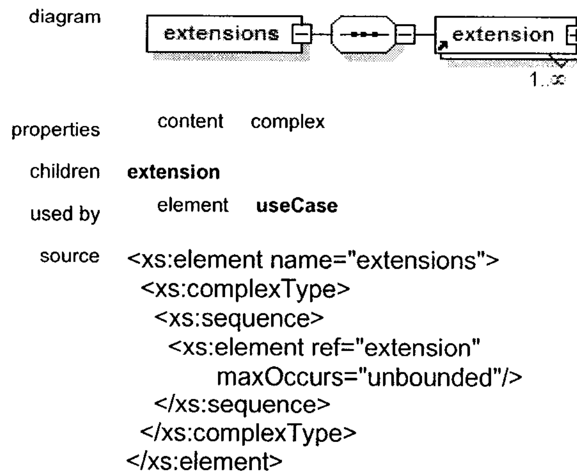


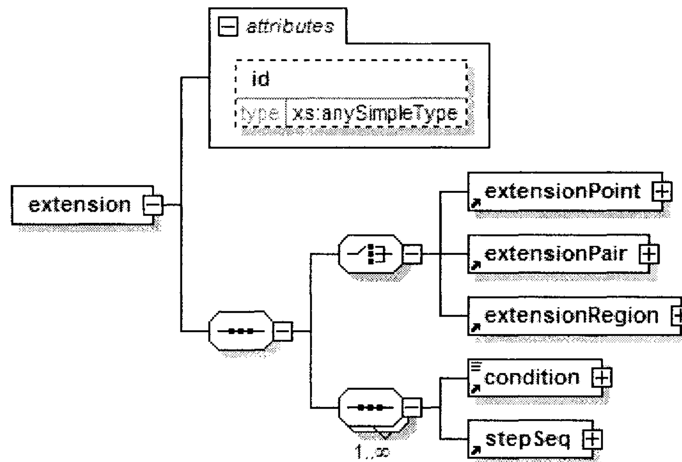
diagram	
properties	content complex
children	**extension**
used by	element **useCase**
source	

```
<xs:element name="extensions">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="extension"
            maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Figure 26. Structure of the extensions

Extension

diagram



properties	content	complex		
children	**extensionPoint extensionPair extensionRegion condition stepSeq**			
used by	element	**extensions**		
attributes	Name	Type	Use	Default
	id	**xs:anySimpleType**		

source

```
<xs:element name="extension">
  <xs:complexType>
    <xs:sequence>
      <xs:choice>
        <xs:element ref="extensionPoint"/>
        <xs:element ref="extensionPair"/>
        <xs:element ref="extensionRegion"/>
      </xs:choice>
      <xs:sequence maxOccurs="unbounded">
        <xs:element ref="condition"/>
        <xs:element ref="stepSeq"/>
      </xs:sequence>
    </xs:sequence>
    <xs:attribute name="id" type="xs:anySimpleType"/>
  </xs:complexType>
</xs:element>
```

Figure 27. Structure of the extension