Refactoring Use Case Models on Episodes

Wei Yu

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of Requirements for the Degree of Master of

Computer Science

Concordia University

Montreal, Quebec, Canada

March, 2004

© Wei Yu, 2004

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisisitons et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-91150-0
Our file Notre référence
ISBN: 0-612-91150-0

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

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

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

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

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

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

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

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

Canada

Abstract

Refactoring Use Case Models on Episodes

Yu Wei

Use case models are used to capture functionality requirements of a system. Use cases can be described in term of the episodes, or subtasks, that are performed. An episode model captures the organization and relationships of the episodes in a use case.

Refactoring is an approach to reorganize the internal structure of models in order to improve them or extend them in some way. Refactorings are behavior-preserving transformations of the models.

This thesis looks at refactoring of use case models based on the information captured in episode models. We present a metamodel for the use case and the episode model in order to make precise the syntax of the models and explain the informal semantics of the models. We detail several refactoring rules for use case refactoring, including their verification of the behavior-preserving property. We also present a case study based on the Video Store System.

Acknowledgements

I would like to express my sincere thanks to my supervisor, Dr. Gregory Butler. This thesis would not be where it is today without his guidance, encouragement, and dedication. Dr. Butler is a valuable asset to the teaching and researching profession, I could not have asked for a better supervisor. I would also like to thank Ke Xing Rui for his valuable dedication and suggestions. Thanks to my team-members and friends, Jian Xu, and RengHong Luo for their participation, and contributions. Special thanks also to my parents and siblings for their support and understating. Thanks also go out to my best friend, Vaillant, for his encouragements and concerns. I would also like to take this chance to thank my darling, Jun Li, for his support and love.

Dedication

To my parents, JinMing Yu and XinYu Yang

Table of Contents

LIST OF FIGURES IX

INTRODUCTION 1

1.1 PROBLEM 1

1.2 WORK 2

1.3 CONTRIBUTIONS 3

1.4 ORGANIZATION OF THESIS 4

BACKGROUND 6

2.1 REFACTORING 6

2.2 CASCADED REFACTORING 7

2.3 USE CASE METAMODEL 8

2.3.1 Regnell's use case metamodel 9

2.3.2 Rui's use case metamodel 11

2.3.3 Metz's refined use case model 16

REFINED METAMODEL AND ITS SPECIFICATION 18

3.1 METAMODEL AT ENVIRONMENT LEVEL 18

3.1.1 Use Case Model 18

3.1.2 Well-formedness Rules for Use Case Model 20

3.2 METAMODEL AT STRUCTURE LEVEL 23

3.2.1 Episode Model 23

3.2.2 Well-formedness Rules for Episode Model 24

3.2.3 Specification for Episode Model ... *26*

REFACTORING RULES ... 29

4.1 REFACTORING PROCESS ... 29

4.2 REFACTORING USE CASE MODEL ON EPISODES ... 30

4.3 REFACTORING RULES ... 32

4.3.1 Inclusion_generation refactoring ... *32*

4.3.2 Extension_generation refactoring ... *35*

4.3.3 Equivalence_generation refactoring ... *39*

4.3.4 Reuse_usecase refactoring ... *43*

4.3.5 Delete_usecase refactoring ... *45*

4.3.6 Generalization_generation refactoring ... *46*

4.3.7 Precedence_generation refactoring ... *53*

4.3.8 Inclusion_mergence refactoring ... *56*

4.3.9 Extension_mergence refactoring ... *58*

4.3.10 Precedence_mergence refactoring ... *60*

4.4 OTHER REFACTORING RULES ... 62

A CASE STUDY ... 63

5.1 INTRODUCTION ... 63

5.1.1 System description ... *63*

5.1.2 Actor characteristics ... *64*

5.2 USE CASE AND EPISODE MODELS ... 66

5.2.1 The use case diagram for VSS ... *66*

5.2.2 Episode models ... *66*

5.3 USE CASE REFACTORINGS 70

CONCLUSION 81

6.1 ACHIEVEMENTS OF THIS THESIS 81

6.2 LIMITATIONS 82

6.3 FUTURE WORK 82

REFERENCES 84

List of Figures

Figure 2.1 Regnell's use case metamodel 9

Figure 2.2 Rui's use case metamodel 11

Figure 2.3 Use case relationship metamodel 12

Figure 2.4 Actor relationship metamodel 14

Figure 2.5 Task relationship metamodel 15

Figure 2.6 Goal relationship metamodel 15

Figure 2.7 Use case model in UML 1.4 16

Figure 2.8 Metz's refined use case model 17

Figure 3.1 Use case model 19

Figure 3.2 Episode model 24

Figure 3.3 Episode tree for the use case "Withdrawing Cash" 28

Figure 4.1 Refactoring process for use case modeling 30

Figure 4.2 Inclusion_generation refactoring 34

Figure 4.3 Use case "Withdrawing cash" with its episode model 35

Figure 4.4 Use case diagram after applying "inclusion_generation refactoring" 35

Figure 4.5 Extension_generation refactoring 37

Figure 4.6 Use case "Paying bill" with its episode tree 38

Figure 4.7 Use case "Paying bill" and its extending use case "Register bill" 39

Figure 4.8 Equivalence_generation refactoring by the developer 41

Figure 4.9 Equivalence_generation refactoring by episode trees 42

Figure 4.10 Use cases with "equivalence" relationship 42

Figure 4.11 Reuse_usecase refactoring 44

Figure 4.12 Use case diagram after applying “reuse_usecase refactoring” 44

Figure 4.13 Delete_usecase refactoring 46

Figure 4.14 Use case diagram after applying “delete_usecase refactoring” 46

Figure 4.15 Generalization_generation refactoring for use cases 50

Figure 4.16 Use case diagram with generalization relationship 51

Figure 4.17 Generalization_generation refactoring for actors 53

Figure 4.18 Precedence_generation refactoring. 55

Figure 4.19 Use case diagram after applying “Precedence_generation refactoring” 56

Figure 4.20 Inclusion_mergence refactoring. 58

Figure 4.21 Extension_mergence refactoring 60

Figure 4.22 Precedence_mergence refactoring 61

Figure 5.1 Use case diagram for the VSS 66

Figure 5.2 Use case diagram after step 1 70

Figure 5.3 Use case diagram for “Web Video Inquiry” after step 2 71

Figure 5.4 Use case diagram after step 3 72

Figure 5.5 Use case diagram after applying “reuse_usecase refactoring” 73

Figure 5.6 Use case diagram for “Customer Maintenance” 74

Figure 5.7 Use case diagram for “Video Maintenance” 76

Figure 5.8 Use case diagram for the VSS after applying refactoring rules 77

Chapter 1

Introduction

1.1 Problem

When developing a system, one of the foremost things is to define its requirements. Since the concept of use case was introduced by Ivar Jacbson [Jacobson92], it has been widely used to capture the functionality requirements of a system. However, obtaining well-organized use case models is not easy because the following reasons:

- Requirements are complicated for a large-scale system. The use case model often contains redundancy when attempting to capture the complete set of behaviors.
- New requirements are captured during the elaborating of use cases and therefore new use cases are added to the system.
- Requirements are evolving. New behaviors are added and existing behaviors may change. Deleting, updating, and adding use cases happen frequently.

Over time, the original design of a use case model is damaged and therefore the cost of changing is escalating. To get well-organized use case models, refactoring gets more and more attention from system analysts.

"Refactoring, as a concept, is a kind of reorganization. It implies equivalence; the beginning and ending products must be functionally identical. Refactoring is a good thing because complex expressions are typically built from simpler, more gradable components. Refactoring either exposes those simpler components or reduces them to the more efficient complex expression." - [Wiki04]

"In software, refactoring is a way to restructure software to make design insights more explicit, to develop frameworks and extract reusable components, to clarify the software architecture, and prepare to make additions easier. Refactoring can help you leverage your past investment, reduce duplication and streamline a program." – Bill Opdyke [Opdyke00]

Refactoring improves the structure of software yet does not change its behavior. Therefore, it is also called behavior preserving transformations. How to obtain well-organized use case models through refactoring technique is one of the popular research topics in software analysis today.

1.2 Work

The purpose of the thesis is to apply the refactoring to use case modeling. Since refactoring as a concept has not previously been applied to use case modeling, there is no clear principle of what to do and how to do it. Through the project, we tried to clarify the refactoring rules used in use case models. Given a case study, we demonstrate how to refactor use case models step by step. With the refactoring, we restructure the use case composition, rearrange their relationships, and reduce the redundancy.

With Dr. Butler and his PHD student Rui's guidance, we developed a use case refactoring tool, which includes several models' editors and implements the refactoring rules that are provided by Rui; we also demonstrated a case study based on the Automated Teller Machine (ATM) with the tool's support. For the *use case refactoring tool*, Jian Xu was responsible for the use case model editor, he implemented the refactoring rules that are related to the use case model. RenHong Luo was responsible for the task and goal models editors, he implemented the refactoring rules that are related to the task and goal models. I, the author of the thesis, was responsible for the episode and event models editors, I implemented the refactoring rules that are related to the episode and event models.

More over, the author refined the use case metamodel at the environment and structure levels; also, I introduced some new use case refactoring rules based on the information captured by episodes; finally, I presented a case study of the Video Store System (VSS) to demonstrate how the refactoring changes the structure of use case models.

1.3 Contributions

The *use case refactoring* is a team-work project. The author made the following contributions for the project.

The use case metamodel at the environment and structure levels are refined based on the analysis of Regnell and Rui's use case metamodel and Metz's contribution for refining use case model. More constraints are applied and are given in OCL (Object Constraint Language). The episode semantics are also explained in informal text.

To explain refactoring in use case modeling, the author defines several use case refactoring rules to change the organization of use case models based on the information captured in episode models. The rules are presented in detail, including motivation, definition, and verification, with an example based on the Automatic Teller Machine (ATM) to demonstrate how the rule is applied in practice. These rules focus on use case composition and decomposition, and the reorganization of their relationships.

To facilitate the use case refactoring, we developed a tool. The tool provides not only editors for building use case models at three levels, but also various refactorings according to the refactoring rules provided by Rui. The author was responsible for programming the use case models at the structure and event levels. First of all, I participated in the discussion of the refactoring rules provided by Rui. Afterward, I implemented an episode model editor and an event model editor using JAVA. Meanwhile, I also defined XML schema files to standardize the format of the episode and event models for storage. Next, I implemented Rui's refactoring rules that related to the structure and event levels. Most of these rules are used to change use case models at the environment level. Finally, a case study based on the Automatic Teller Machine (ATM) is demonstrated using the tool.

To demonstrate how the refactoring rules introduced in this thesis are applied to change the organization of use case models, the Video Store System (VSS) is analyzed and used as a case study for the refactorings. Through this case study, some of the refactoring rules are applied and the transformation of use case models is presented step by step.

1.4 Organization of Thesis

The thesis is organized into 5 chapters.

- *Chapter 2:* This chapter introduces the background and basic knowledge used in the thesis. The previous work of refactoring, such as cascaded refactoring, is presented. Regnell's and Rui's use case metamodels, which are the basis of our use case tool, are analyzed in detail.

- *Chapter 3:* In this chapter, we refine the metamodel of use case at the environment and structure levels. The formal syntax of the refined metamodel is given using OCL. The informal semantic of the metamodel is also explained.
- *Chapter 4:* In this chapter, we discuss the refactorings applied in use case modeling. Several use case refactoring rules based on episodes are explained in detail from different perspectives: motivation, definition, verification, and example.
- *Chapter5:* This chapter presents a case study based on the Video Store System (VSS). Through this case study, we demonstrate step-by-step how the refactoring rules introduced in chapter 4 are applied to change the organization of use case models.

Chapter 2

Background

This chapter introduces the basic knowledge that are used in our refactoring project. The concept of refactoring is introduced. Especially, the cascaded refactoring for framework evolution is explained. Next, the previous work on the use case metamodel, such as Regnell and Rui's, which forms the basis of our refined metamodel is discussed in detail. Furthermore, Metz's contribution of refining the use case model is explained briefly.

2.1 Refactoring

The concept of refactoring comes from mathematics when we transform an expression into an equivalent expression with factors. The factors express the same statement yet with clearer organization. Therefore, refactoring is a kind of reorganization that implies equivalence [Wiki03] [Mens04].

In software, refactoring is the process of changing a software system in such way that it does not alter the external behavior of the code yet improves its internal structure [Fowler99]. Therefore, it is also called restructuring or "behavior preserving transformation". It is one of the core element of lightweight or agile methodologies such as eXtreme Programming. It is also an effective strategy for planning the rejuvenation of legacy software [Miller01].

Refactoring has been successfully used in source code reorganization in software; it makes code clearer, simpler, and elegant. In 1992, William F.Opdyke first published 26 lower-level and 3 high level refactorings in his PHD thesis [Opdyke92]. In 1999, Lance Tokuda evolved object-oriented designs with refactorings[Tokuda99]. To make refactoring more practical, Donald Bradley Roberts developed the Refactoring Browser for Smalltalk in 1999 [Roberts99]. Martin Fowler focused on the refactoring process and provided principles for practicing refactorings in his book, *Refactoring: Improving the Design of Existing Code* [Fowler99], to make refactoring more practical. Dr. Butler extended refactoring to various models of a framework, called cascaded refactoring [Butler01], which is introduced in next section.

2.2 Cascaded Refactoring

Cascaded refactoring for the development and evolution of application frameworks is a hybrid approach combining the modeling aspects of top-down domain engineering approaches and the iterative, refactoring approaches of the bottom-up objected community [Butler01]. It extends refactoring to a set of models that describe a framework. The set of models in cascaded refactoring include [Butler01]:

- A feature model organizing common and variable features;
- A use case model of requirements including variation points;
- An architectural model with hotspots;
- A design showing collaborations and the overlapping of roles from design patterns onto the hotspots;
- The source code, with classes, hooks, and templates.

Cascaded refactoring is the process of sequentially refactoring of each model, one after the other, where the restructuring of the previous model becomes the rationale or constraints for the current refactoring of a model. Cascaded refactoring relates to a set of refactorings across the set of models. Particularly, the refactorings for use case models include [Butler01]:

- create_abstract_actor identifies two actors a_1 and a_2 with a common super-actor a as their parent.
- create_abstract_usecase identifies two use cases u_1 and u_2 as specializations of a common super use case u.
- merge_actors identifies two actors a_1 and a_2 as a common actor a.
- merge_behaviours identifies two use cases u_1 and u_2 as a common use case u.
- split_behavior distributes a set of scenarios for a use case u across two new use cases u_1 and u_2.
- split_actor identifies the special cases a_1 and a_2 of an actor a.
- make_episode_usecase takes a use case u with an episode e and create a new use case u_1 with behaviour precisely that of the episode e, a relationship link u includes u_1 is added.
- make_scenario_usecase takes a use case u with a scenario s and creates a new use case u_1 with behavior precisely that of the scenario s, a relationship link u uncludes u_1 is added.

Based on the above refactorings for use case models, Rui suggested more refactoring rules in his PHD proposal, which forms the basis of our refactoring tool.

2.3 Use case metamodel

This section discusses the use case metamodel provided by Regnell first; followed by Rui's modified metamodel, which forms the basis of our use case refactoring tool. In addition, Metz's contribution to refine the use case metamodel is explained briefly.

2.3.1 Regnell's use case metamodel

Use cases have been widely used in functionality requirements gathering and eliciting since Ivar Jacobson first introduced the concept [Jacbson92]. A use case metamodel is the basis for the software development that starts from use case recognition. Figure 2.1 shows Regnell's use case metamodel [Regnell99].

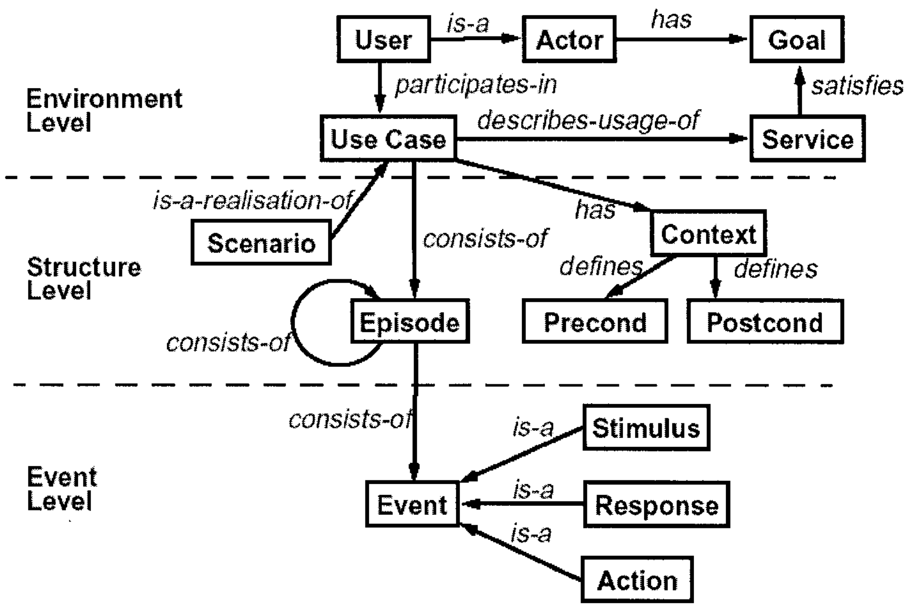


Figure 2.1 Regnell's use case metamodel

The use case metamodel includes three levels, which reflect different abstractions respectively. At the environment level, a use case is related to the entities that are internal to the intended system. At the structure level, the internal structure of a use case is revealed together with its different variants and parts. At the event level, individual events are characterized at lower abstraction level [Rui01].

At the environment level, the key elements are use cases and actors. The use case is used to define the behavior of a system without revealing the entity's internal structure. Each use case specifies a sequence of activities that the entity can perform, interacting with actors of the entity. An actor defines a coherent set of roles that users of an entity can play when interacting with the entity. An actor may be considered to play a separate role with regard to each use case with which it communicates [UML01]. Other elements include users, services, and goals. A user is an instance of an actor. A service is a package of functional entities (features) offered to the users in order to satisfy one or more goals that the users have. A goal is an objective that users have when using the services of a target system. Goals are often used to categorize users into actors. The elements for each level are introduced in following paragraphs.

At the structure level, the key elements are scenarios, episodes and contexts. A scenario is a specific and bound realization of a use case described as a sequence of a finite number of events with linear time order [Rui01]. A use case may be divided into coherent parts, called episodes. These coherent episodes reveal the internal structure of a use case, and describe the functionalities of a use case. A context defines the scope of a use case, including its preconditions and postconditions. Preconditions describe the state or status of the system before a use case executes; postconditions describe the status of the system as a result of the use case completing [Armour01].

At the event level, the key elements are events. There are three types of events: stimulus, action, and response. A stimulus is the message from users to the target system. A response is the message from the target system to users. An action is the target system intrinsic event which is atomic in the sense that there is no communication between the target system and the users that participate in the use case [Rui01]. The difference between an event and an episode is that an event takes no duration whereas an episode does.

2.3.2 Rui's use case metamodel

Rui presented a refined metamodel in his PHD proposal, which is shown in Figure 2.2 [Rui01].

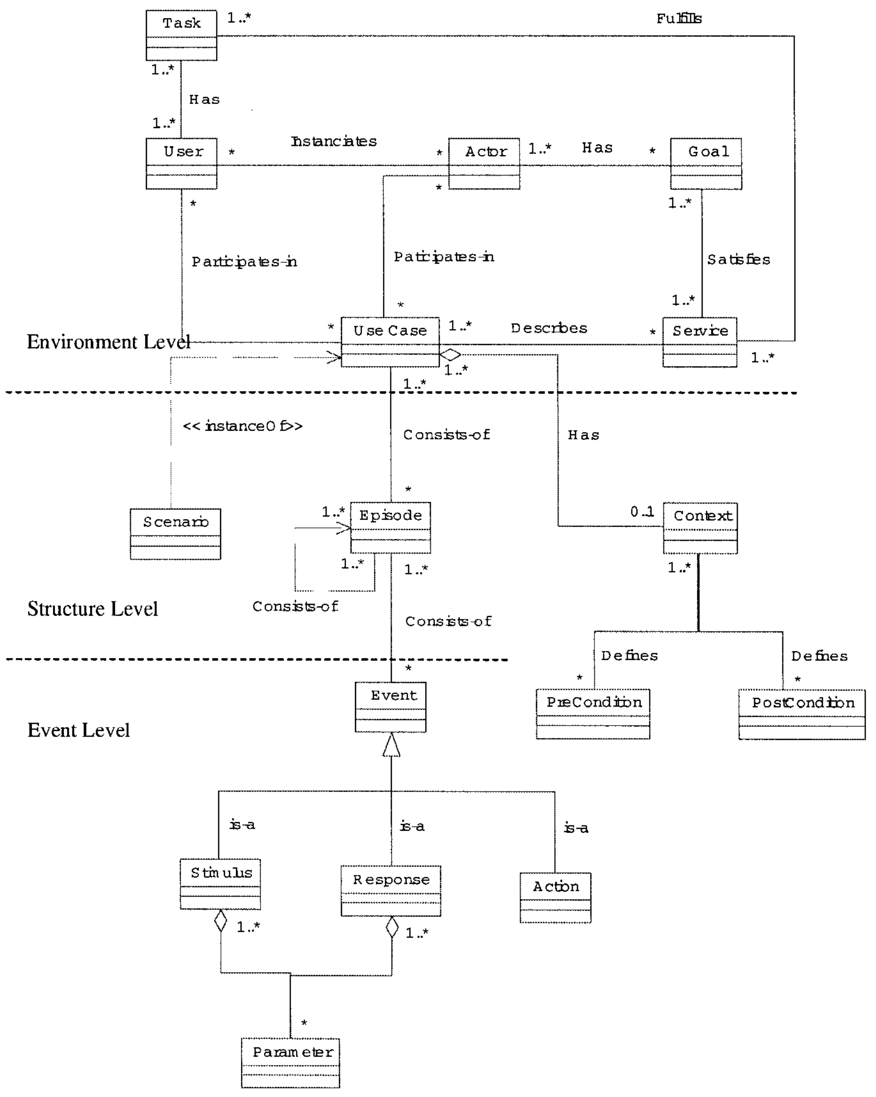


Figure 2.2 Rui's use case metamodel

Rui extended the metamodel with the concept of task. A task describes what the user will do with the software system; it provides better support for user interface design [Rui01].

In addition, Rui introduced more relationships for use cases. He also defined actor relationships, task relationships, and goal relationships, which people can use to organize a hierarchical actor structure, task structure, and goal structure. The metamodels for use case relationship, actor relationship, task relationship, and goal relationship are introduced in the following.

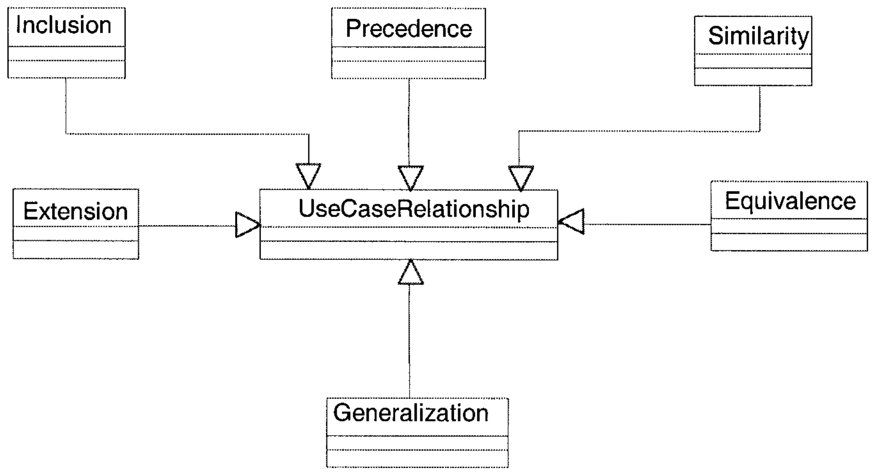


Figure 2.3 Use case relationship metamodel

The use case relationship metamodel is shown in Figure 2.3. The relationships among use cases are explained as follows:

- Inclusion

An inclusion relationship between two use cases means that the behavior defined in the target use case is included at one location in the sequence of behavior performed by an instance of the base use case. It specifies that one use case explicitly incorporate the behavior of another at the given point. When one use case instance reaches the location where the behavior of another use case is to be included, it performs all the behavior described by the included use case and then continues according to its original use case. One use case may be included in several other use cases and one use case may include several other use cases. The included use case may not be dependent on the base use case. In that sense the included use case represents encapsulated behavior that may easily be reused in several use cases. Moreover, the base use case may only be dependent on the results of performing the included behavior and not on structure, like Attributes and Associations, of the included use case [Rui01].

- Extension

An extension relationship between use cases specifies that one use case extends the behavior of another at the given extension point. One use case extends another by introducing alternative or exceptional processes. It defines that instances of a use case may be augmented with some additional behavior defined in an extending use case. The extension relationship contains a condition and references a sequence of extension points in the target use case [Rui01].

- Generalization

A generalization relationship between use cases implies that the child use case contains all the attributes, sequences of behavior, inclusion points, and extension points defined in the parent use case, and participate in all relationships of the parent use case. The child use case may also define new behavior sequences, as well as add additional behavior into and specialize existing behavior of the inherited ones [Rui01].

- Similarity

A similarity relationship between use cases means that one use case corresponds to or is similar to or resembles another in some unspecified ways. Similarity is a relationship often noted early in use case modeling. It provides a way to carry forward insight about relationships among use case even when the exact nature of the relationship is not yet clear [Rui01].

- Equivalence

An equivalence relationship between use cases specifies that one use case serves as an alias for another use case. Equivalence flags those cases where a single definition can cover what are, from the user's perspective, two or more different intensions. It makes it easier to validate the model with users and customers while also assuring that only one design will be developed [Rui01].

- Precedence

A precedence relationship between use cases means that one use case is sequenced (appended) to the behavior of the preceding use case [Rui01]. Precedence is a relationship that exposes the chronological order of use cases. It is a special case of inclusion relationship.

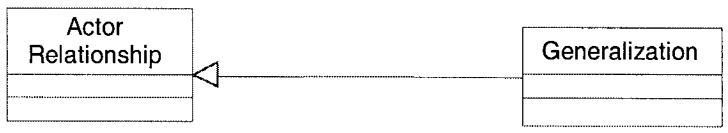


Figure 2.4 Actor relationship metamodel

The metamodel for actor relationship is shown in Figure 2.4. If two or more actors may have commonalities, i.e. communicate with the same set of use cases in the same way, the commonality is expressed with generalizations to another (possibly abstract) actor, which models the common role(s). This means that the child actor will be able to play the same roles as the parent actor, i.e. communicate with the same set of use cases, as the parent actor, in the same way [Rui01].

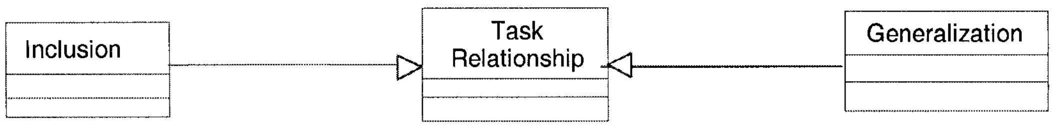


Figure 2.5 Task relationship metamodel

The metamodel for task relationship is shown in Figure 2.5. The relationships among tasks are explained as follows.

- Inclusion

A task may contain other tasks, which are called subtasks. A subtask may be included in several tasks [Rui01].

- Generalization

Two or more tasks may have commonalities, i.e. contain the same set of subtasks. The commonality is expressed with generalizations to another task, which models the common task(s) [Rui01].

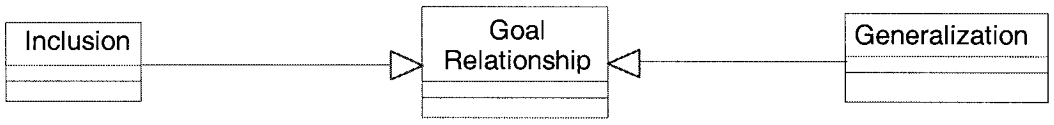


Figure 2.6 Goal relationship metamodel

The metamodel for goal relationship is shown in Figure 2.6. The relationships among goals are explained as follows.

- Inclusion

A goal may contain other goals, which are called subgoals. A subgoal may be included in several other goals and a goal may include several subgoals [Rui01].

- Generalization

Two or more goals may have commonalities, i.e. contain the same set of subgoals. The commonality is expressed with generalizations to another goal, which models the common goal(s) [Rui01].

2.3.3 Metz's refined use case model

UML is accepted by most software developers. The use case model in UML 1.4 is shown in Figure 2.7.

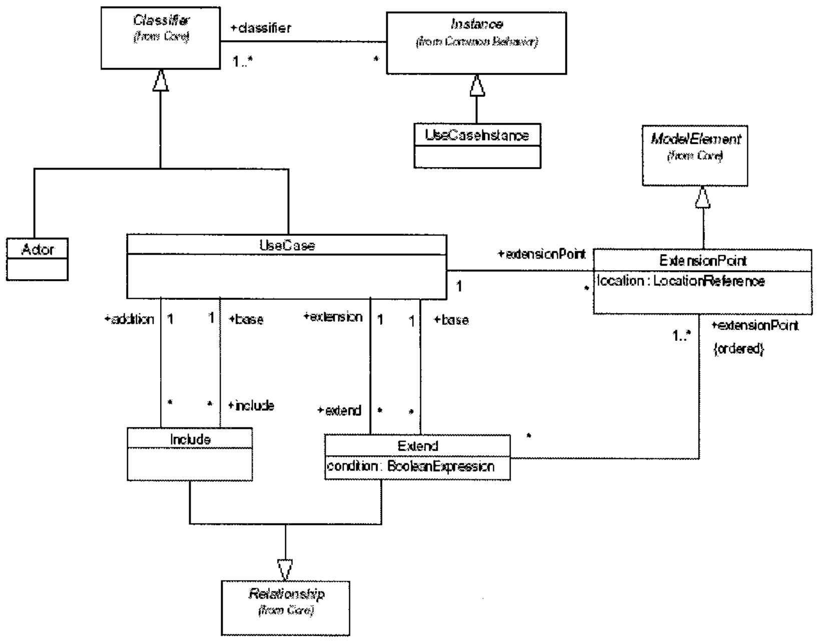


Figure 2.7 Use case model in UML 1.4

To remove inconsistencies caused by the notion of use case interleaving [Metz01a], Metz [Metz01b] suggested a refined use case model, shown in Figure 2.8.

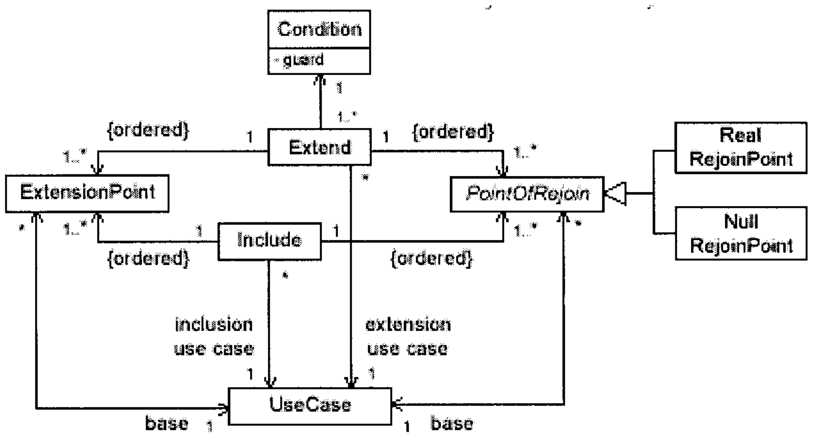


Figure 2.8 Metz's refined use case model

In Metz's refined use case model, the attribute *condition* of *Extend* is removed from *Extend* and reintroduced as an explicit class *Condition*. Furthermore, a metamodel element *PointOfRejoin* with two subtypes *RealRejoinPoint* and *NullRejoinPoint* is introduced to specify a reference to the single return location within the base interaction sequence. An extend relationship will have an associated *RealRejoinPoint* if it represents an alternative part; an extend-relationship will have a *NullRejoinPoint* if it represents either an alternative history or a use case exception.

Inspired by Metz, we refined the use case metamodel at the environment and structure levels, namely use case model and episode model, which are discussed in the next chapter.

Chapter 3

Refined metamodel and its specification

This chapter introduces the refined use case metamodel. The use case model, which is a refined metamodel at the environment level, is given first. Meanwhile, more constraints for its syntax are explained in OCL. Afterward, the episode model, which is a refined metamodel at structural level, is presented. Similarly, more constraints for its syntax are given in OCL. Its informal semantics is also explained briefly.

3.1 Metamodel at environment Level

3.1.1 Use Case Model

Use cases, actors, and relationships among them form a use case model. In addition to inclusion, extension, and generalization relationships between use cases, more relationships, such as precedence, similarity, equivalence, are introduced to our refined use case model. The refined use case model is shown in Figure3.1.

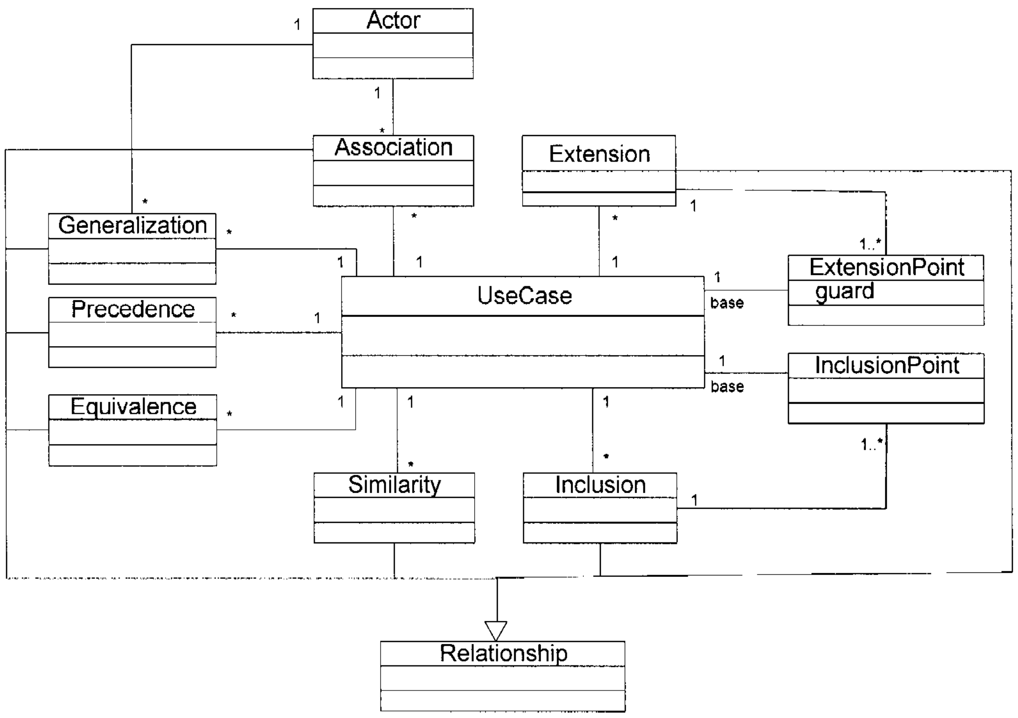


Figure 3.1 Use case model

In Figure 3.1, *PointOfRejoin* is removed from our model since use case interleaving is harmful and should be avoided [Simons99] [Metz01b]. In addition, since the inclusion relationship implies that the behavior in included use case will be inserted into the base use case, the inserting point must be clear in the base use case. However, it is hard to figure out the actual position of an inclusion point without discovering the base use case's internal structure of behavior. Therefore, the insertion points are associated with not only the base use case and the inclusion relationship, but also the episode model of the base use case.

The extension relationship implies that the behavior defined in the extending use case will be inserted into the base use case based on a certain condition. Similarly, the extension points are associated with not only the base use case and the extension relationship, but also the episode model of the base use case.

3.1.2 Well-formedness Rules for Use Case Model

A diagram is typically not refined enough to provide all the relevant aspects of a specification. Therefore, it is necessary to depict the additional constraints about objects in the model. OCL (Object Constraint Language) is a formal language that remains easy to read and write. It is developed to supply a gap between a natural language and a formal language when they are used to describe constraints. It is also widely used in UML [UML01] [Larman98] [Booch99].

For our use case metamodel, OCL is used to define its constraints. Moreover, we can define stereotypes that are specific for our model using UML's extension mechanism.

The following well-formedness rules apply to the use case model.

Context *Actor* inv:

1) *Actor* can only have *Association* relationship to *UseCase*

self.associations->forAll(a | a.connection->size = 2 and

a.allConnections ->exists(r | r.type.oclIsKindOf(Actor)) and

a.allConnections ->exists(r | r.type.oclIsKindOf(UseCase)))

2) The name of an *Actor* must be unique within a use case model

self.allInstance->forAll(a_1*,* a_2 *|* a_1*.name =* a_2*.name implies* $a_1 = a_2$*)*

Context *UseCase* inv:

3) The name of a UseCase must be unique within a use case model.

self.allInstance->forAll(u_1*,* u_2 *|* u_1*.name =* u_2*.name implies* $u_1 = u_2$*)*

4) *Usecase* can only have binary *Extensions.*

self.extensions -> forAll(a | a.connection ->size = 2)

5) *Usecase* can only have binary *inclusions.*

self.inclusions -> forAll(a | a.connection ->size = 2)

6) *Usecase* can only have binary *Equivalences.*

self.equivalences -> forAll(a | a.connection ->size = 2)

7) *Usecase* can only have binary *Similarities.*

self.similarities -> forAll(a | a.connection ->size = 2)

8) *Usecases* can only have binary *Generalizaions.*

self.generalizations -> forAll(a | a.connection ->size = 2)

9) The names of the *ExtensionPoints* must be unique within a *UseCase.*

self.allExtensionPoints -> forAll(x, y | x.name = y.name implies x = y)

10) The names of the *InclusionPoints* must be unique within a *UseCase.*

self.allInclusionPoints -> forAll(x, y | x.name = y.name implies x = y)

<u>Context Extension inv:</u>

11) *Extension* can connect only two *UseCases* within a use case model.

self.allInstances->forAll(a | a.connection->size = 2 and a.allConnections->forAll(r | r.type.oclIsKindOf(UseCase)))

12) The *ExtensionPoints* must be included in set of *ExtensionPoint* in the target *UseCase.*

self.base.allExtensionPoints -> includesAll(self.extensionPoint)

<u>Context *Inclusion* inv:</u>

13) *Inclusion* can connect only two *UseCases* within a use case model.

self.allInstances->forAll(a | a.connection->size = 2 and a.allConnections->forAll(r | r.type.oclIsKindOf(UseCase)))

14) The *InclusionPoints* must be included in set of *InclusionPoint* in the target *UseCase.*

self.base.allInclusionPoints -> includesAll(self.inclusionPoint)

<u>Context *Equivalence* inv:</u>

15) *Equivalence* can connect only two *UseCases* within a use case model.

self.allInstances->forAll(a | a.connection->size = 2 and a.allConnections->forAll(r | r.type.oclIsKindOf(UseCase)))

<u>Context *Similarity* inv:</u>

16) *Similarity* can connect only two *UseCases* within a use case model.

self.allInstances->forAll(a | a.connection->size = 2 and a.allConnections->forAll(r | r.type.oclIsKindOf(UseCase)))

<u>Context *Generalization* inv:</u>

17) *Generaliztion* can connect only two *UseCases* or two *Actors* within a use case model.

self.allInstances->forAll(a | a.connection->size = 2 and (a.allConnections->forAll(r | r.type.oclIsKindOf(UseCase))) or (a.allConnections->forAll(r | r.type.oclIsKindOf(Actor))))

18) *Association* can connect only *Actor* and *UseCase* within a use case model.

self.allInstances->forAll(a | a.connection->size = 2 and a.allConnections->exists(r | r.type.oclIsKindOf(Actor)) and a.allConnections->exists(r | r.type.oclIsKindOf(UseCase)))

<u>Context *ExtensionPoint* inv:</u>

19) The name of *ExtensionPoint* must be unique.

self.allInstance->forAll(p_1, p_2 | p_1.name = p_2.name implies $p_1 = p_2$)

20) The name can not be empty

not self.name = ''

Context *InclusionPoint* inv:

21) The name of *InclusionPoint* must be unique.

self.allInstance->forAll(p_1, p_2 | p_1.name = p_2.name implies $p_1 = p_2$)

22) The name can not be empty

not self.name = ''

3.2 Metamodel at Structure level

3.2.1 Episode Model

A use case can be described in terms of episodes. Each episode represents a subtask, or a part of the dialogues in the scenario that perform the subtask [Butler01]. A high-level Message Sequence Charts (MSCs) can be used to depict a use case and its episodes, while an operator expresses *sequence*, *alternative*, *iteration*, or *parallel* relationships among episodes [Regnell96].

The behavior of a use case is described in an episode model. Elements in the episode model include primitive episodes and composite episodes. With the inclusion and extension relationship between use cases, the inserting point related to inclusion or extension relationship must be clarified within the episode model. Therefore, another element, namely pseudo episode, is introduced in the episode model to indicate the inclusion or extension points for inclusion or extension relationship between use cases. For a composite episode, it takes *sequence, alternation, iteration, and parallel* as its operator, episodes as its parameters.

Based on the above analysis, the refined episode model is shown in Figure 3.2.

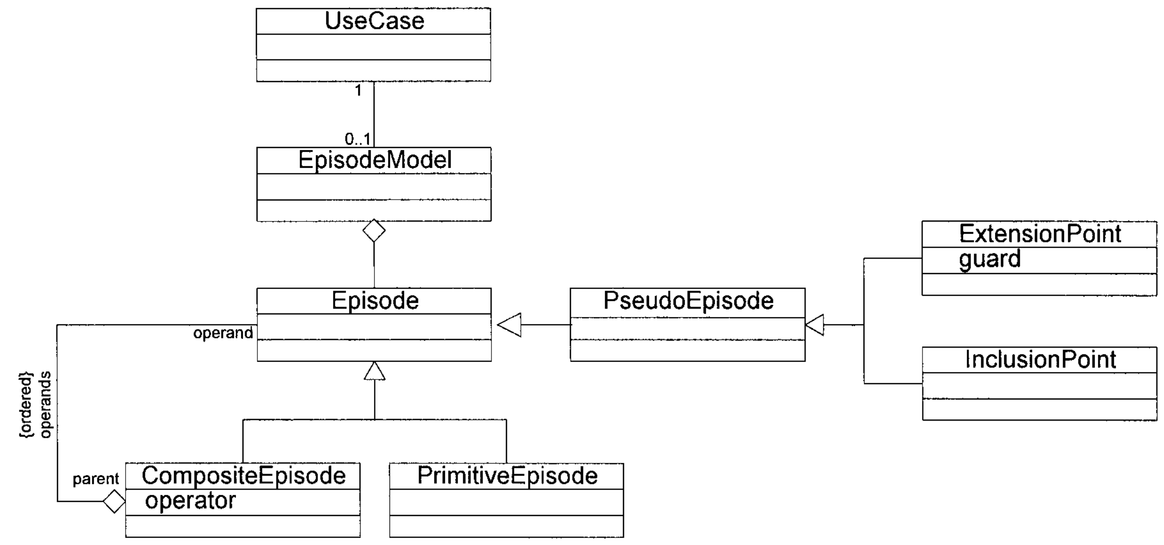


Figure 3.2 Episode model

3.2.2 Well-formedness Rules for Episode Model

We use OCL to explain the episode model's constraints. The following well-formedness rules apply to the episode model.

Context *UseCase* inv:

1) A use case can be refined using an episode model.

Self.EpisodeModel->size() = 0 or

Self.EpisodeModel->size() = 1

Context *EpisodeModel* inv:

2) An episode model can be represented as a tree, only one composite episode acts as the root for the episode tree.

self.Episode.CompositeEpisode.allInstances-> isUnique(e| e.isRoot = true)

3) An episode model refines only one use case.

self.UseCase->size() = 1

Context *Episode* inv:

4) An Episode is specified as a composite episode, primitive episode, or pseudo episode in an episode model.

self.EpisodeModel.oclType.elements(forAll(m |

m.stereotype = PrimitiveEpisode or m.stereotype = CompositeEpisode or

m.stereotype = PseudoEpisode))

Context *CompositeEpisode* inv:

5) There is only one episode tree in an episode model and the root is a composite episode.

self.allInstances->isUnique(e| e.isRoot = true)

6) A composite episode is a root node in an episode tree when its parent set is empty.

self.parent->isEmpty implies self.isRoot = true;

7) There are 4 kinds of operators in a composite episode: sequence, alternative, iteration, and parallel.

self.operator.type: enum {sequence, alternative, iteration, parallel}

8) A composite episode consists of one or more composite episodes, primitive episodes, or pseudo episodes.

let c = self.child

c(forAll (ce| ce.typeOfEpisode = CompositeEpisode or ce.typeOfEpisode =

PrimitiveEpisode or ce.typeOfEpisode = PseudoEpisode)

Context *PrimitiveEpisode* inv:

9) The name of *PrimitiveEpisode* can not be empty

not self.name = ''

10) The name of *PrimitiveEpisode* must be unique.

self.allInstance->forAll($p_1, p_2 \mid p_1.name = p_2.name$ *implies* $p_1 = p_2$*)*

11) A primitive episode is an atomic episode.

self.type = atomic

Context *PseudoEpisode* inv:

12) The name of *PseudoEpisode* can not be empty

not self.name = ''

13) The name of *PseudoEpisode* must be unique.

self.allInstance->forAll($p_1, p_2 \mid p_1.name = p_2.name$ *implies* $p_1 = p_2$*)*

3.2.3 Specification for Episode Model

An episode model consists of various episodes. An episode that defines a basic activity for implementing the behavior of a use case is considered a primitive episode. Episodes with their operator, that define a complex activity, are considered a composite episode. The operators allowed are *sequence, alternation, iteration, and parallel.* An episode that indicates an insert point for the inclusion or extension relationship between use cases is considered as a pseudo episode.

Based on the above analysis, episodes are classified into 6 categories:

- *Sequence* episode: a composite episode. The child episodes are executed one-by-one in order; this relationship is used to describe sequential activities in a use case.

- *Alternative* episode: a composite episode. Only one of the child episodes may be executed based on the condition; the condition is defined in the alternative episode. This relationship is used to describe alternative course in a use case.
- *Iteration* episode: a composite episode. The child episodes are executed multiple times according to the parameter that is defined in the iteration episode. This relationship is used to describe repetitive activities in a use case.
- *Parallel* episode: a composite episode. The child episodes have no time order constraints for each other. In other words, they can be executed in any order. This relationship is used to describe the parallel activities in a use case.
- *Primitive* episode: an atomic episode. It is the fundamental element in an episode model; it can not be divided.
- *Pseudo* episode: a special episode. It is used to indicate the inclusion point or extension point in the base use case when an inclusion or extension relationship exists between the base use case and the related use case.

Adopting the tree structure, it is easy to represent the relationship between a composite episode and its child episodes. A parent node in the tree represents a composite episode. A leaf node in the tree represents an atomic episode. The type of a parent node that could be *sequence*, *alternative*, *iteration*, or *parallel*, implies the relationship among its child nodes. For example, in the ATM (Automatic Teller Machine) system, the episode tree of a use case *Withdrawing Cash* is shown in Figure3.3.

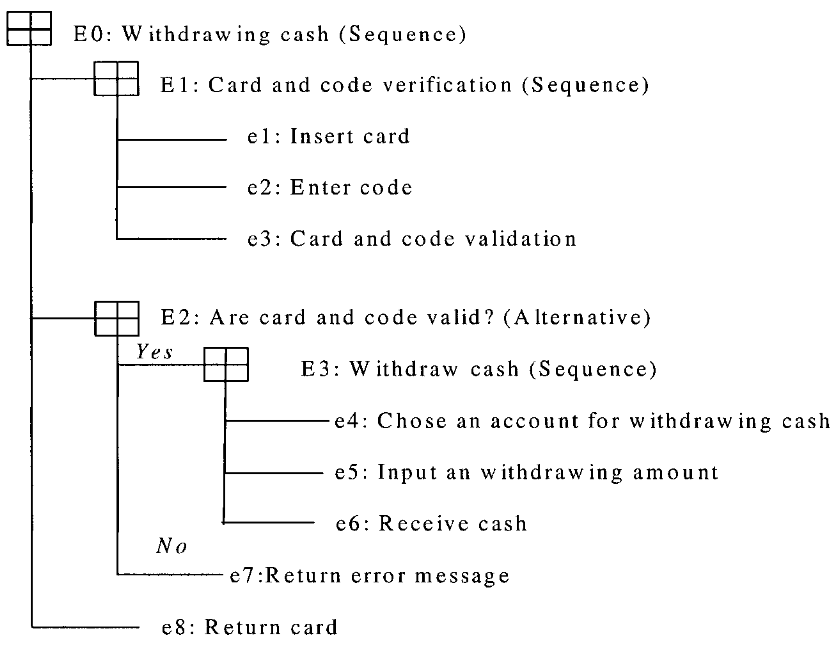

E0: Withdrawing cash (Sequence)
E1: Card and code verification (Sequence)
e1: Insert card
e2: Enter code
e3: Card and code validation
E2: Are card and code valid? (Alternative)
Yes
E3: Withdraw cash (Sequence)
e4: Chose an account for withdrawing cash
e5: Input an withdrawing amount
e6: Receive cash
No
e7:Return error message
e8: Return card

Figure 3.3 Episode tree for the use case "Withdrawing Cash"

In Figure 3.3, E0 is a sequence episode; its three children episodes E1, E2, and e8 are executed one by one. E2 is an alternative episode; either E3 or e7 will be executed. E2 and E3 are also sequence episodes.

Episode composition and decomposition not only make a use case understandable and be of manageable size, but also improve reusability. Moreover, an episode model can be easily translated into an indented textual notation by a depth-first traversal of the tree.

Chapter 4

Refactoring Rules

This chapter introduces use case refactoring rules based on episode models. The refactoring process for use case refactoring is first presented. Next, the general refactoring is explained briefly. Finally, several refactoring rules are selected and explained in detail from motivation, definition, and verification perspectives. In addition, an example following each rule is given to demonstrate how the rule works in practice.

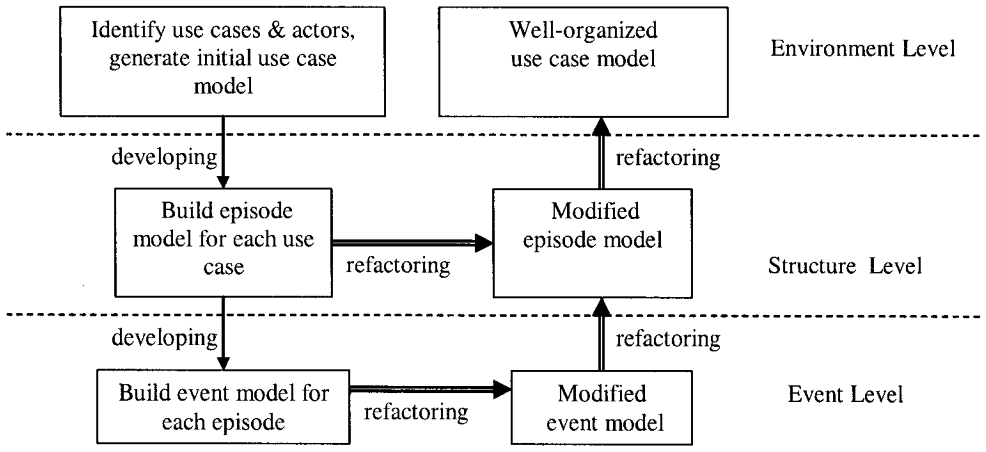
To simplify, the episode tree is described in indented textual notation. Dewey Decimal system is used to index each sentence. Abbreviations for the type of composite episodes are: "Seq" stands for "Sequence", "Alt" stands for "Alternation", "Ite" stands for "Iteration", "Par" stands for "Parallel".

4.1 Refactoring process

As the use case metamodel shows, a use case is depicted at three levels: environment, structure, and event level. The environment level shows high-level specifications of functionalities of a system, which are described by use cases, actors and the relationships among them. The structure level depicts the specific behaviors and interactions that take place between the actor and system within the scope of the use case in terms of episodes. The event level represents a lower abstraction level where the interacting messages and events are characterized.

What use case refactoring does is to choose the correct levels of abstraction and granularity, identify commonality, eliminate redundancy and reuse functionality.

In general, a top-down development and bottom-up refactoring process is used. We start from the environment level, and gradually reveal the details of each use case at the structure level or go further to the event level to get the initial model of a project. Next, starting from the event level or structure level, we combine commonalities and abandon redundancies to get clean and "slim" structure level diagrams or environment level diagrams. The process is shown in Figure 4.1. It is an iterative and incremental process.



Identify use cases & actors, generate initial use case model
Well-organized use case model
Environment Level
developing
refactoring
Build episode model for each use case
refactoring
Modified episode model
Structure Level
developing
refactoring
Build event model for each episode
refactoring
Modified event model
Event Level

Figure 4.1 Refactoring process for use case modeling

4.2 Refactoring use case model on episodes

One of the significant parts of building the use case model involves breaking up use cases in search of simpler ones and factoring out common behavior and variant paths [Scott02] [Armour01]. In our research, we start from the episode model to discover common behaviors and therefore break up use cases. Such refactoring will result in the following relationships between use cases:

- **Inclusion**

With an inclusion relationship, one use case explicitly includes the behavior of another use case at a specified point within its main course of activities. The included use case may be connected with one or more base use cases. The inclusion mechanism is used to factor out the common behavior, that would appear within multiple use cases.

- **Extension**

With an extension relationship, a base use case implicitly includes the behavior of another use case at one or more specified points under certain conditions. The extension mechanism is used to factor out the optional behavior that having several steps associated with it.

- **Generalization**

Generalization works the same way for use cases as it does for classes: a parent use case defines behavior that its children can inherit, and the children can add to or override that behavior. The generalization mechanism is used to factor out the similar structure of behavior in two or more use cases. These use cases share similar structure of the behavior that may be abstracted into their parent use case. Meanwhile, the common primitive episodes are removed to their parent use case and then are inherited by them.

- **Similarity**

With a similarity relationship, the related use case must share similar structure of behavior or primitive episodes. However, the exact nature of the relationship is not clear yet. A similarity is created temporarily and will be eventually deleted when the nature of relationship is clear. The similarity mechanism is used to carry forward the potential relationship that could be clarified in the future.

- **Equivalence**

With an equivalence relationship, the related use cases are behaviorally equivalent. The equivalence mechanism is used to describe two equivalent use cases that exist for different users.

- **Precedence**

With a precedence relationship, the preceding use case executes prior to the base use case. A precedence relationship is generally factored out from an inclusion relationship. For example, "ATM withdrawal" can be broken into two smaller use cases: "ID verification" and "Withdraw money". We can say "Withdraw money" use case includes "ID verification" use case. However, such relationship is too vague to precisely render that the use case "ID verification" must occur prior to the use case "Withdraw money". In such situation, a precedence relationship can describe the chronological order more precisely than an inclusion relationship.

The precedence mechanism is used to factor out the chronological order between use cases. It is also a special case inclusion relationship but it is more effective in discovering chronological order between use cases than an inclusion relationship.

4.3 Refactoring rules

The use case refactoring rules that are based on the information captured in episode models are explained one by one in following sub-sections.

4.3.1 Inclusion_generation refactoring

Motivation

Constructing an inclusion relationship is one of the efficient ways to break up use cases into smaller use cases. When the episode model of a use case is too complicated, the best way is to break up the episode tree and therefore a new use case with a sub-tree as its episode tree is separated from the base use case. Meanwhile, an inclusion relationship from the base use case to the new use case is generated.

The specific process is: take a sub-tree from the episode tree of the base use case; construct a new use case with the sub-tree; generate an inclusion relationship from the base use case to the new use case; substitute the sub-tree in the base use case by a pseudo episode, which indicates the inclusion point for the inclusion relationship. The refactoring helps to reduce granularity, eliminate redundancy, and improve the reusability.

Definition

Preconditions:

- A use case *u* with its episode tree *t*;
- The sub-tree t_i, which describes relative independent part of functionality of *u*.
- None of t_i's ancestors is an alternative composite episode (this condition guarantees that the included use case must take place when the base use case is executed).

Parameters:

- *u*: the use case that is to be broken up;
- *t*: the episode tree of *u*;
- t_i: the sub-tree of *t*;

Postconditions:

- The sub-tree t_i of *t* is replaced by a node t_i', where t_i' is a pseudo episode that indicates an inclusion point.
- A new use case *u'* is generated; its episode tree is t_i;

- A new inclusion relationship from *u* to *u'* is generated.

Verification

Let us analyze the semantics of the inclusion relationship. Use case *u* includes use case *u'* means that *u'* is part of *u*. In other words, *u'* implements part of functionality of *u*. When *u* executes, *u'* will be invoked at a specific point that the pseudo episode t_i' defines.

From the view of the episode tree, the specific point is defined in t_i', which is the node that substitutes t_i. When refactoring, t_i is taken off from *t* and becomes the episode tree of *u'*. Following the inclusion relationship, the episode tree of use case *u'* is invoked and t_i' is substituted as what it was before the refactoring. Therefore, the inclusion_generation refactoring is a behavior preserving transformation. The refactoring is shown as Figure 4.2.

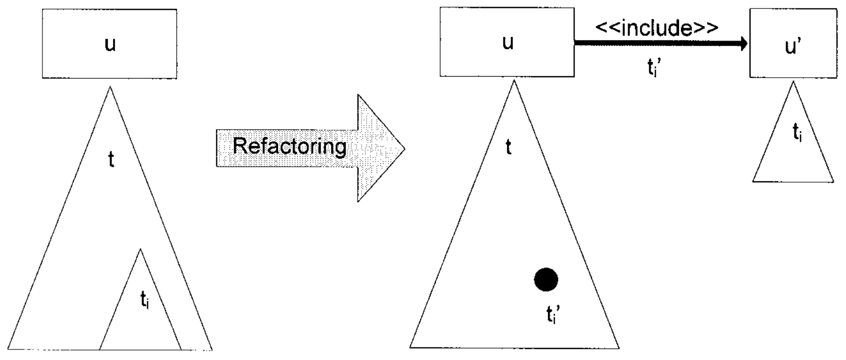


Figure 4.2 Inclusion_generation refactoring

Example

In the ATM system, suppose there is a use case "Withdrawing cash" with its episode tree, shown in Figure 4.3.

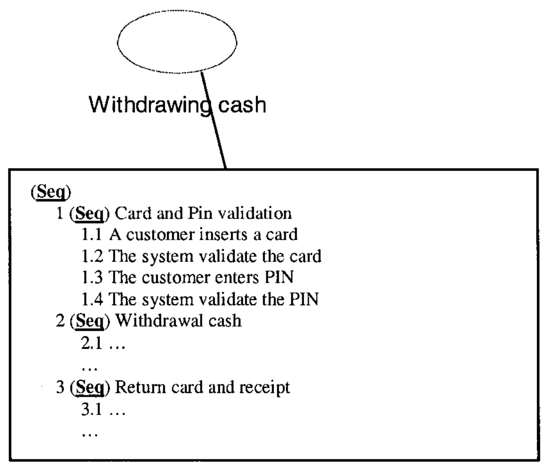

Withdrawing cash
(Seq)
1 (Seq) Card and Pin validation
1.1 A customer inserts a card
1.2 The system validate the card
1.3 The customer enters PIN
1.4 The system validate the PIN
2 (Seq) Withdrawal cash
2.1 ...
...
3 (Seq) Return card and receipt
3.1 ...
...

Figure 4.3 Use case "Withdrawing cash" with its episode model

From the episode tree, we notice that the sub-tree "Card and Pin validation" is complicated and can be extracted as a new use case. The use case diagram after applying "inclusion_generation refactoring" is shown in Figure 4.4.

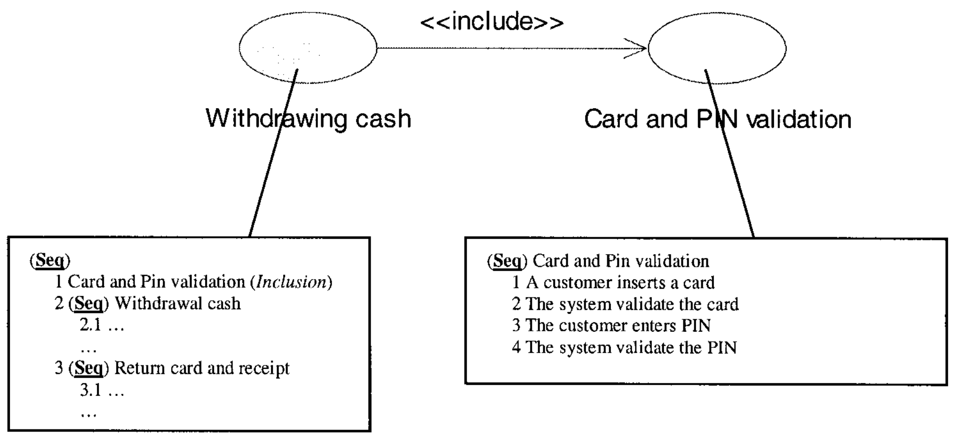

<<include>>
Withdrawing cash
Card and PIN validation
(Seq)
1 Card and Pin validation (Inclusion)
2 (Seq) Withdrawal cash
2.1 ...
...
3 (Seq) Return card and receipt
3.1 ...
...
(Seq) Card and Pin validation
1 A customer inserts a card
2 The system validate the card
3 The customer enters PIN
4 The system validate the PIN

Figure 4.4 Use case diagram after applying "inclusion_generation refactoring"

4.3.2 Extension_generation refactoring

Motivation

An extension relationship specifies one use case extends the behavior of another at the given extension point. One use case extends another by introducing alternative processes.

The relationship defines that instances of a use case may be parameterized with some additional behavior defined in the extending use case.. When an alternative course is complicated enough to construct a new use case, an extension relationship is introduced and the use case is broken up into two smaller use cases.

The specific process is: take a sub-tree of an alternative composite episode from the episode tree of the base use case; construct a new use case with the sub-tree; generate an extension relationship from the new use case to the base use case; substitute the sub-tree in the base use case by a pseudo episode, which indicates the extension point for the extension relationship. This refactoring helps to reduce complexity and improve reusability.

Definition

Preconditions:

- A use case *u* with its episode tree *t*,
- A sub-tree t_i, which is an alternative course of the use case *u*. In other words, t_i's parent is an "alternative" composite episode (this guarantees that the extending use case may take place based on the specific condition that is defined in the "alternative" parent in the base use case).

Parameters:

- *u*: the base use case;
- *t*: the episode tree of *u*;
- t_i: the sub-tree of *t*, which is an alternative course of the use case *u*;

Postconditions:

- A new use case *u'* with its episode tree t_i is generated;

- An extension relationship from *u'* to *u* is generated.
- The sub-tree t_i in the use case *u* is replaced by a pseudo episode named t_i', which indicates the extension point in the base use case *u*.

Verification

Let us analyze the semantics of the extension relationship. Use case *u'* extends use case *u* means that *u'* will take place only if the condition of extension is *true* and then *u'* will be inserted into the extension point where t_i' defines.

From the view of the episode tree, the extending use case is inserted into the location of t_i' when the extension condition is *true*. In other words, the node t_i' is substituted by the tree of t_i, and therefore the episode tree of the base use case *u* is restored to its original form. When the extension condition is *false*, t_i' remains as a pseudo episode in the base use case. Since the branch of t_i' will never be reached when the extension condition is *false*, the behavior of the use case even is not affected whether or not t_i' is substituted by t_i. Therefore, the extension_generation refactoring is a behavior preserving transformation. The refactoring is shown in Figure 4.5.

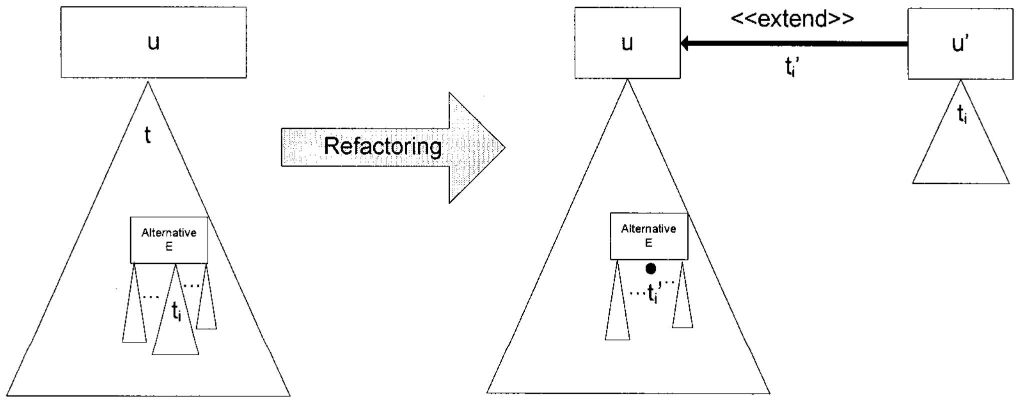
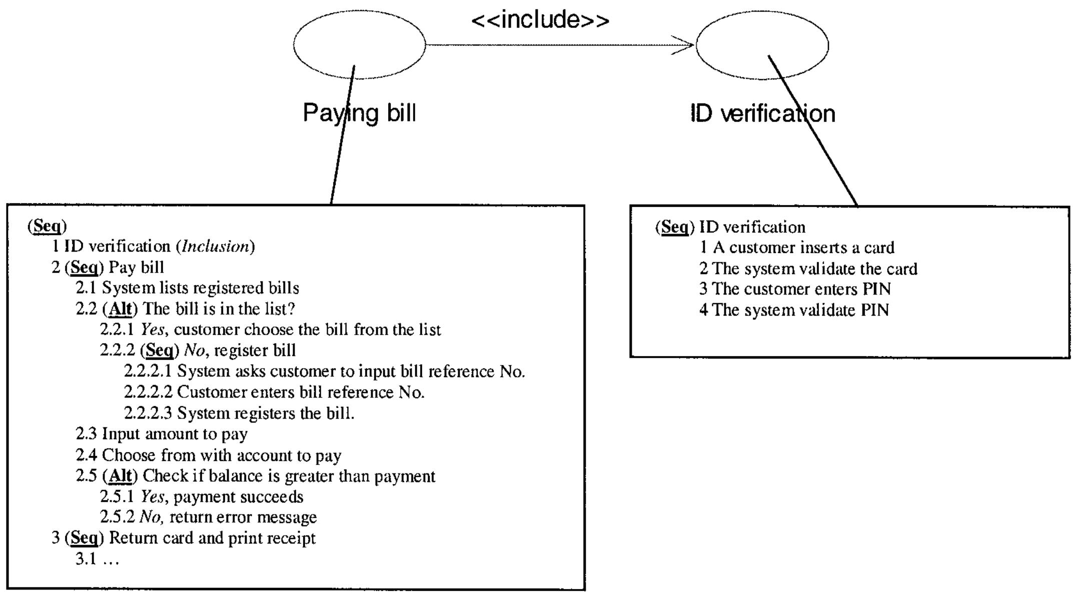


Figure 4.5 Extension_generation refactoring

Example

Suppose there is a use case "Paying bill" in the ATM system. The episode tree is shown in Figure 4.6.



```
(Seq)
  1 ID verification (Inclusion)
  2 (Seq) Pay bill
    2.1 System lists registered bills
    2.2 (Alt) The bill is in the list?
      2.2.1 Yes, customer choose the bill from the list
      2.2.2 (Seq) No, register bill
        2.2.2.1 System asks customer to input bill reference No.
        2.2.2.2 Customer enters bill reference No.
        2.2.2.3 System registers the bill.
    2.3 Input amount to pay
    2.4 Choose from with account to pay
    2.5 (Alt) Check if balance is greater than payment
      2.5.1 Yes, payment succeeds
      2.5.2 No, return error message
  3 (Seq) Return card and print receipt
    3.1 ...
```

```
(Seq) ID verification
    1 A customer inserts a card
    2 The system validate the card
    3 The customer enters PIN
    4 The system validate PIN
```

Figure 4.6 Use case "Paying bill" with its episode tree

From Figure 4.6, we find out that "register bill" is an alternative course of "Paying bill" and it represents a relative independent functionality. Therefore the sub-tree can be removed from the use case "Paying bill" and we can create a new use case called "register bill". The relationship from the use case "register bill" to the base use case "Paying bill" should be "extend" since "register bill" takes place based on condition "the bill is **not** in the list" (in other words, the bill is not registered). The use case diagram with its episode trees after applying "extension_generation refactoring" is shown in Figure 4.7.

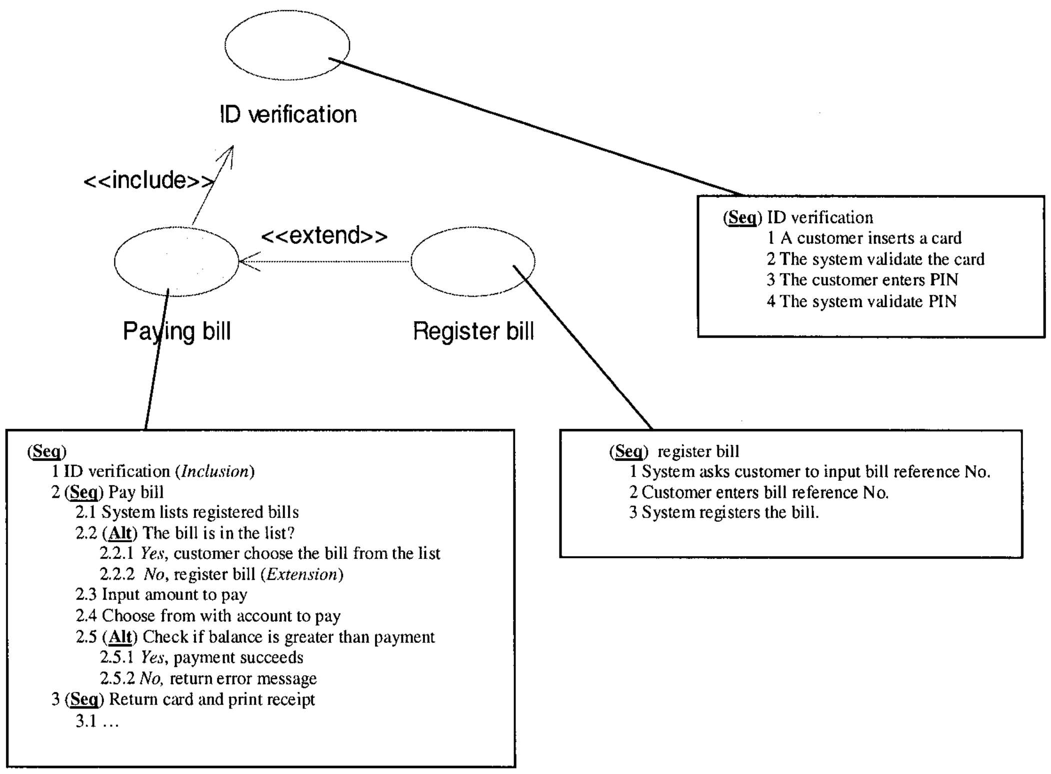

ID verification
<<include>>
<<extend>>
Paying bill
Register bill
(Seq) ID verification
1 A customer inserts a card
2 The system validate the card
3 The customer enters PIN
4 The system validate PIN
(Seq)
1 ID verification (Inclusion)
2 (Seq) Pay bill
2.1 System lists registered bills
2.2 (Alt) The bill is in the list?
2.2.1 Yes, customer choose the bill from the list
2.2.2 No, register bill (Extension)
2.3 Input amount to pay
2.4 Choose from with account to pay
2.5 (Alt) Check if balance is greater than payment
2.5.1 Yes, payment succeeds
2.5.2 No, return error message
3 (Seq) Return card and print receipt
3.1 ...
(Seq) register bill
1 System asks customer to input bill reference No.
2 Customer enters bill reference No.
3 System registers the bill.

Figure 4.7 Use case "Paying bill" and its extending use case "Register bill"

4.3.3 Equivalence_generation refactoring

There are two situations for equivalence_generation refactoring: one is generated from the viewpoint of the developer; another is generated from the definition of the episode tree. They are introduced in the following two sections respectively.

4.3.3.1 Equivalence_generation by the developer

Motivation

From the viewpoint of the developer, two use cases may be behaviorally equivalent. In this case, an equivalence relationship is generated based on the developer's consideration. From the view of the episode model, the two use cases should have an equivalent episode tree; therefore, the episode tree is duplicated from the one that has an episode tree to the one that has no an episode tree. If both of the use cases have episode trees, the equivalence condition is verified before generating the equivalence relationship between them; this belongs to the second situation.

Definition

Preconditions:

- Two use cases u_1 and u_2 are behaviorally equuivalent from the viewpoint of the developer;
- At least one of them has no episode tree defined.

Parameters:

- u_1, u_2: the equivalent use cases

Postconditions:

- An equivalence relationship is generated between u_1 and u_2;
- If one of them has episode tree defined, the episode tree is duplicated to the one that has no episode tree.

Verification

Since an equivalence relationship does not participate in any behavior of the use cases, the behavior of the use cases have no change after the refactoring. Therefore, the equivalence_generation refactoring is a behavior preserving transformation. The refactoring is shown in Figure 4.8.

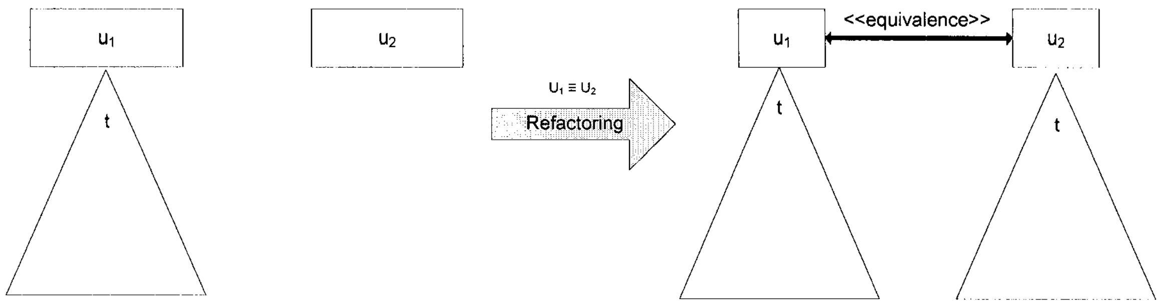


Figure 4.8 Equivalence_generation refactoring by the developer

4.3.3.2 Equivalence_generation refacoring from the definition of the episode tree

Motivation

If two use cases have equivalent episode trees, we conclude that the two use cases are equivalent in behavior. Then an equivalence relationship is suggested between these two use cases.

Definition

Precondition:

- The episode trees of two use cases are behaviorally equivalent.

Parameters:

- u_1, u_2: the use cases that have equivalent episode trees.

Postcondition:

- An equivalence relationship is generated between u_1 and u_2.

Verification

Since an equivalence relationship does not participate in any behavior of the use cases, the behavior of the use cases have no change after the refactoring. Therefore, the equivalence_generation refactoring is a behavior preserving transformation. The refactoring is shown in Figure 4.9.

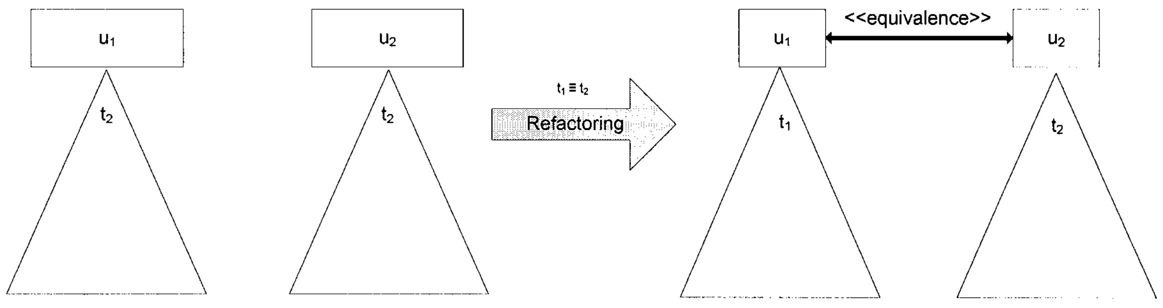


Figure 4.9 Equivalence_generation refactoring by episode trees

Example

Compare Figures 4.4 and 4.6, we find out that the episode trees "Card and PIN validation" and "ID verification" are the same. If the developer is not sure of applying "reuse_usecase refactoring", "equivalence_generation refactoring" can be applied. The use case diagram after applying "equivalence_generation refactoring" is shown in Figure 4.10.

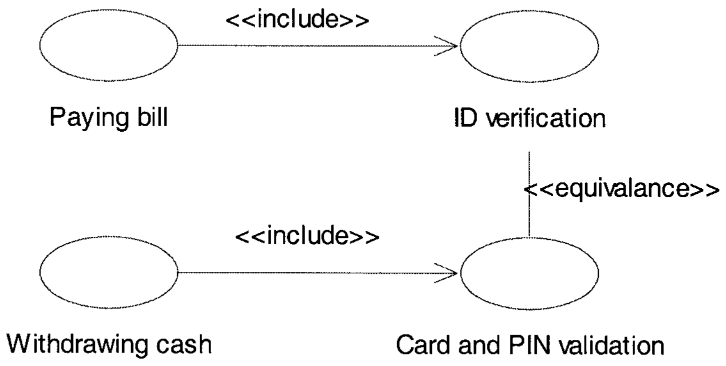


Figure 4.10 Use cases with "equivalence" relationship

4.3.4 Reuse_usecase refactoring

Motivation

When two use cases are equivalent in behavior, two solutions will be suggested: one is to generate equivalence relationship; another is to reuse. The first is introduced in equivalence_generation refactoring. The latter one "reuse" is to replace one with the other. The specific process is: the replacing use case takes over all relationships of the replaced use case. Obviously, the refactoring helps to improve the reusability.

Definition

Precondition:

- The episode trees of two use cases are equivalent.

Parameters:

- u_1, u_2: use cases.

Postcondition:

- All relationships with u_2 are taken over by u_1.

Verification

Since u_1 is behaviorally equivalent to u_2, all behaviors implemented by u_2 will be implemented equally by u_1 after the relationships with u_2 are taken over by u_1. Therefore, the reuse_usecase refactoring is a behavior preserving transformation. The refactoring is shown in Figure 4.11.

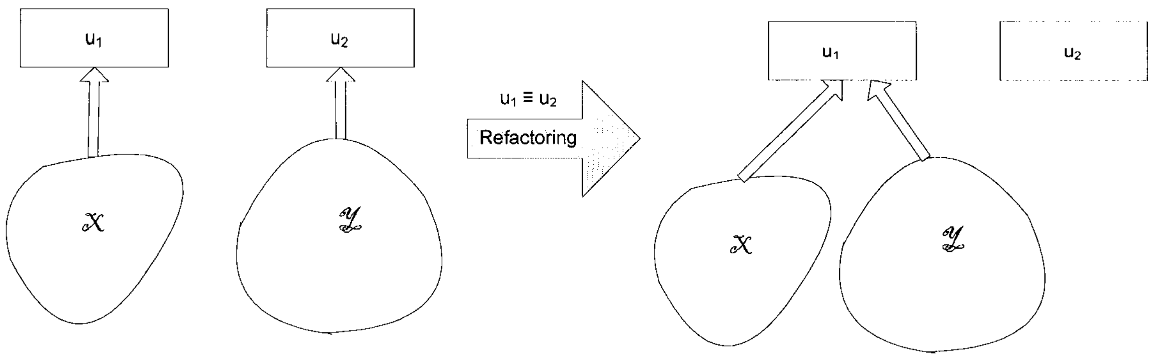


Figure 4.11 Reuse_usecase refactoring

Example

From Figure 4.10, we notice that use case "Card and PIN validation" is equivalent to "ID verification". Therefore, "reuse_usecase refactoring" can be applied to reduce redundancy and inconsistency. The use case diagram after applying "reuse_usecase refactoring" is shown in Figure 4.12.

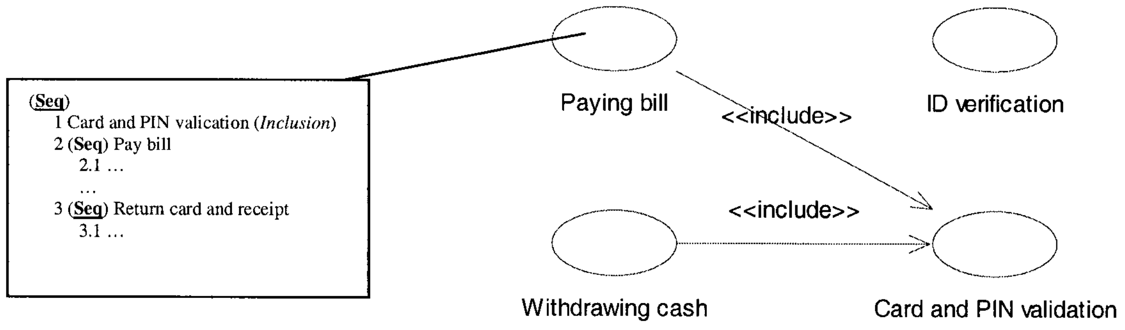


Figure 4.12 Use case diagram after applying "reuse_usecase refactoring"

Note: the inclusion point in the episode tree of "Paying bill" has been changed since the inclusion relationship is changed to "Card and PIN validation". Relationship "equivalence" is omitted since it is from and to the same use case "Card and PIN validation" after applying "reuse_usecase refactoring".

4.3.5 Delete_usecase refactoring

Motivation

When a use case is defined but does not participate in any relationships with other use cases or actors, the use case is considered as a redundant use case in the system. Particularly, more use cases will become redundant after applying reuse_usecase refactoring. These redundant use cases can be deleted from the system. This refactoring helps to reduce redundancy.

Definition

Precondition:

- A use case *u* has no relationships with other use cases or actors.

Parameters:

- *u*: the use case that has no relationships with any other use cases or actors.

Postcondition:

- The use case *u* is deleted from the system.

Verification

Since the redundant use case does not participate in any relationships with other use cases or actors, deleting it has no effect to the behavior of the system. Therefore, the delete_usecase refactoring is a behavioral preserving transformation. The refactoring is shown in Figure 4.13.

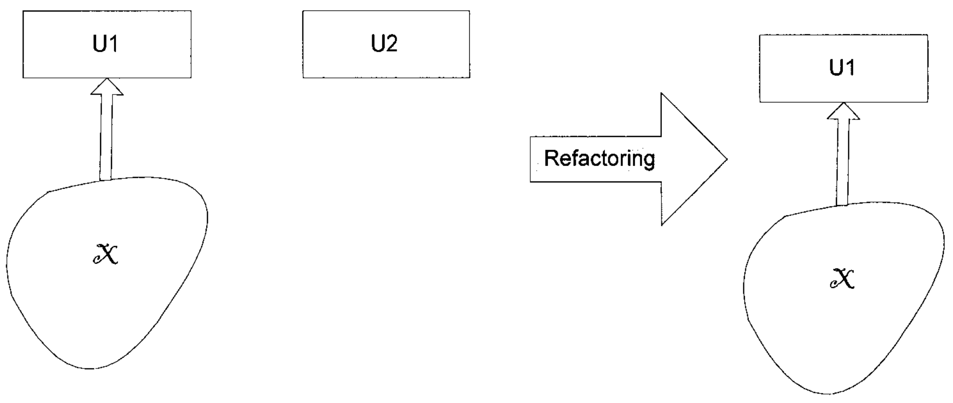


Figure 4.13 Delete_usecase refactoring

Example

From Figure 4.12, the use case "ID verification" is redundant since it does not interact with the system. Therefore it can be deleted from the system to reduce redundancy. The use case diagram after applying "delete_usecase refactoring" is shown in Figure 4.14.

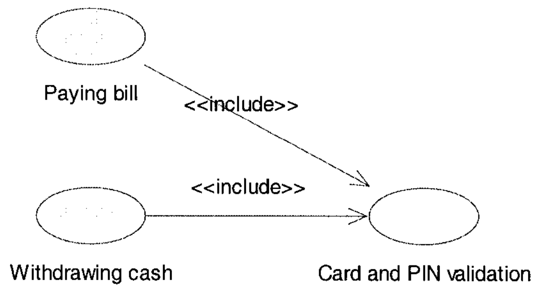


Figure 4.14 Use case diagram after applying "delete_usecase refactoring"

4.3.6 Generalization_generation refactoring

A generalization relationship exists between use cases or actors. They are introduced in the following two sections respectively.

4.3.6.1 Generalization_generation refactoring for use cases

Motivation

A generalization relationship between use cases implies that the child use case contains all the attributes and behavior sequence defined in the parent use case, and participates in all relationships of the parent use case. The child use case may inherit the behaviors of its parent, add new behaviors, and specialize the behaviors inherited from the parent.

Generalization_generation refactoring is suggested when two or more use cases are isomorphic, share common primitive episodes, and there is an episode tree which the episode trees of the use cases are special cases of. The generalization is complicated and it is controversial on how to generate it. To simplify the problem, we only take common primitive episodes as precondition; the rest are judged by the developer.

The specific process is: a new parent use case is generated based on these use cases; the common primitive episodes are removed to the parent use case; generalization relationships from those use cases to the parent use case are generated. The refactoring helps to eliminate redundancy and improve the reusability.

Note: Use cases u_1 and u_2 are isomorphic is explained as follows.

Suppose u_1 has relationships with a set of use cases, denoted as $X = \{x_1, x_2, \dots\}$; u2 has relationships with a set of use cases, denoted as $Y = \{ y_1, y_2, \dots\}$; there is a mapping Φ: X->Y, and $x_i \equiv \Phi(x_i)$. Then we have that: if u1 has relationship ® with x_i, then u_2 has the same relationship with $\Phi(x_i)$; if x_i has relationship with x_j, then $\Phi(x_i)$ has the same relationship with $\Phi(x_j)$.

Definition

Preconditions:

- Use cases u_1 and u_2 are isomorphic;

- The use case u_1 has its episode tree t_1; the use case u_2 has its episode tree t_2. There is an episode tree t, where t_1 and t_2 are special cases of t;
- t_1 and t_2 share common primitive episodes $\{e_1, e_2, ..., e_i\}$;

Parameter:

- u_1, u_2: chosen use cases;

Postconditions:

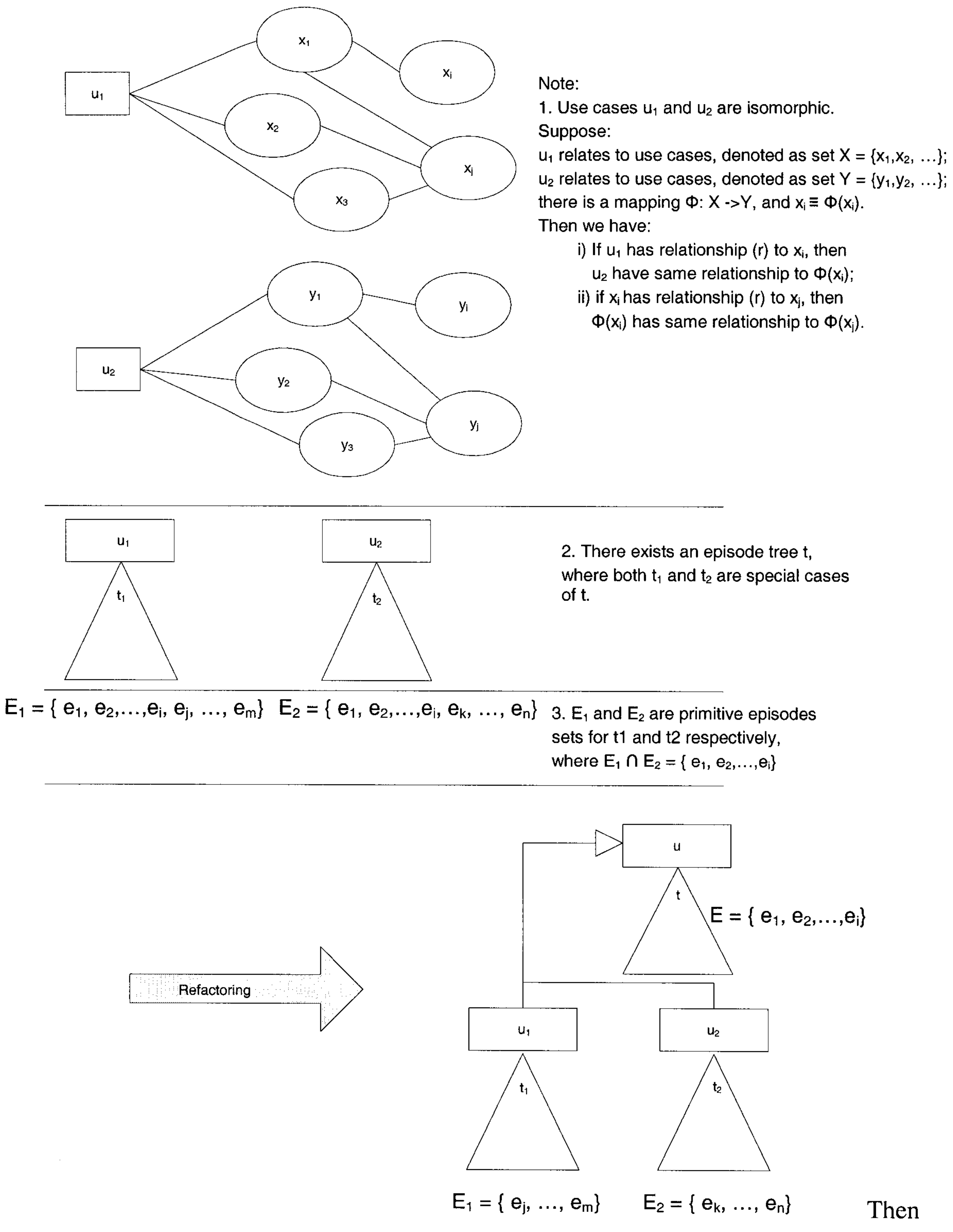
- A new use case u with its episode tree t is generated;
- Generalization relationships from u_1 and u_2 to the parent use case u are generated;
- Primitive episodes $\{e_1, e_2, ..., e_i\}$ are removed from children use cases u_1 and u_2 to the parent use case u;
- Common relationships between $\{u_1, u_2\}$ and other use cases or actors are taken over by u from $\{u_1, u_2\}$.

Verification

Let us analyze the semantics of the generalization relationship. The use case u generalizes use cases $\{u_1, u_2\}$ means that u_1 and u_2 could inherit attributes (primitive episodes) and the behaviors from the use case u, add new behaviors. In addition, u_1 and u_2 could specialize the behaviors inherited from the parent use case u.

From the view of episode model, the common primitive episodes $\{e_1, e_2, ..., e_i\}$ are removed from use cases $\{u_1, u_2\}$ to their parent use case u; then with the generalization relationship, $\{e_1, e_2, ..., e_i\}$ are inherited from the parent use case u. Therefore, the generalization_generation refactoring for use cases is a behavior preserving transformation. The refactoring is shown in Figure 4.15.

Note: the generalization_generation refactoring for use cases is too complicated and needs to be refined since here we only check common primitive episodes as precondition. We leave this for future research.


u_1
x_1
x_2
x_3
x_i
x_j
u_2
y_1
y_2
y_3
y_i
y_j
u_1
t_1
u_2
t_2
$E_1 = \{ e_1, e_2, \ldots, e_i, e_j, \ldots, e_m \}$
$E_2 = \{ e_1, e_2, \ldots, e_i, e_k, \ldots, e_n \}$
Refactoring
u
t
$E = \{ e_1, e_2, \ldots, e_i \}$
u_1
t_1
u_2
t_2
$E_1 = \{ e_j, \ldots, e_m \}$
$E_2 = \{ e_k, \ldots, e_n \}$
Then

Note:
1. Use cases u_1 and u_2 are isomorphic.
Suppose:
u_1 relates to use cases, denoted as set $X = \{x_1, x_2, \ldots\}$;
u_2 relates to use cases, denoted as set $Y = \{y_1, y_2, \ldots\}$;
there is a mapping Φ: X ->Y, and $x_i \equiv \Phi(x_i)$.
Then we have:
i) If u_1 has relationship (r) to x_i, then u_2 have same relationship to $\Phi(x_i)$;
ii) if x_i has relationship (r) to x_j, then $\Phi(x_i)$ has same relationship to $\Phi(x_j)$.

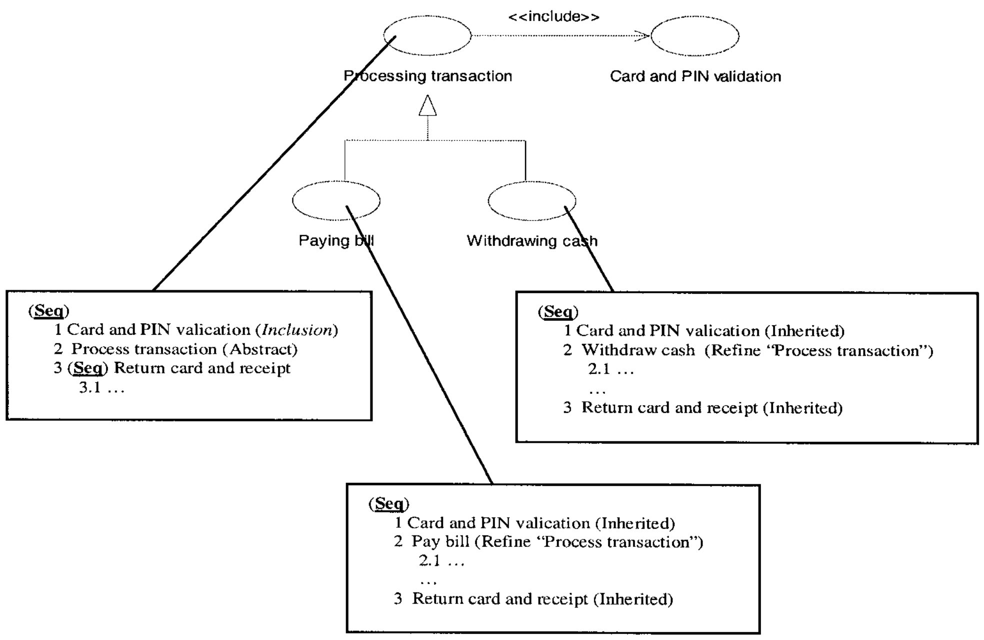
2. There exists an episode tree t, where both t_1 and t_2 are special cases of t.

3. E_1 and E_2 are primitive episodes sets for t1 and t2 respectively, where $E_1 \cap E_2 = \{ e_1, e_2, \ldots, e_i \}$

Figure 4.15 Generalization_generation refactoring for use cases

Example

From the use cases "Paying bill" and "Withdrawing cash", we notice that they have common primitive episodes and belong to the same family. Therefore, generalization_generation refactoring is suggested to reduce redundancy and improve reusability. The use case diagram after applying "generalization_generation refactoring for use cases" is shown in Figure 4.16.



<<include>>
Processing transaction
Card and PIN validation
Paying bill
Withdrawing cash
(Seq)
1 Card and PIN valication (*Inclusion*)
2 Process transaction (Abstract)
3 (Seq) Return card and receipt
3.1 ...
(Seq)
1 Card and PIN valication (Inherited)
2 Withdraw cash (Refine "Process transaction")
2.1 ...
...
3 Return card and receipt (Inherited)
(Seq)
1 Card and PIN valication (Inherited)
2 Pay bill (Refine "Process transaction")
2.1 ...
...
3 Return card and receipt (Inherited)

Figure 4.16 Use case diagram with generalization relationship

4.3.6.2 Generalization_generation refactoring for actors

Motivation

When two or more actors communicate with the same set of use cases in the same way, this commonality can be expressed with generalizations to another actor, which models the common role(s). This means that the child actor will be able to play the same roles as the parent actor, i.e. communicate with the same set of use cases, as the parent actor. The refactoring helps to organize hierarchical structure for actors, and therefore make use case model simpler and clearer.

Definition

Preconditions:

- Actors a_1 and a_2 communicate with the same set of use cases $\{u_1, u_2, \ldots, u_k\}$ in the same way. Suppose the common relationships with $\{u_1, u_2, \ldots, u_k\}$ is $Z = \{r_1, r_2, \ldots, r_k\}$;
- Actors $\{a_1, a_2\}$ belong to the same family.

Parameters:

- a_1, a_2: chosen actors

Postconditions:

- A new actor a is generated;
- Association relationships $\{r_1, r_2, \ldots, r_k\}$ from the actor a to use cases $\{u_1, u_2, \ldots, u_k\}$ are generated. This means that the actor a communicates with the same use cases in the same way as that of actors $\{a_1, a_2\}$
- Generalization relationships from actors a_1 and a_2 to the parent actor a are generated;
- Relationships $\{r_1, r_2, \ldots, r_k\}$ from actors a_1 and a_2 to use cases $\{u_1, u_2, \ldots, u_k\}$ are deleted. These relationships could be inherited from their parent actor a.

Verification

Before the refactoring, there are common relationships $\{r_1, r_2, \ldots, r_k\}$ from actors a_1 and a_2 to use cases $\{u_1, u_2, \ldots, u_k\}$ After the refactoring, these relationships are deleted.

However, a new actor *a* is generated, and it has the same relationships $\{r_1, r_2, ..., r_k\}$ with use cases $\{u_1, u_2, ..., u_k\}$ as that of actors a_1 and a_2. In addition, generalization relationships from actors a_1 and a_2 to the parent actor *a* are generated. According to the definition of generalization between actors, the child actor will be able to play the same roles as the parent actor, i.e. communicate with the same set of use cases, as the parent actor, actors $\{a_1, a_2\}$ communicate with use cases $\{u_1, u_2, ..., u_k\}$ in the same way $\{r_1, r_2, ..., r_k\}$ as their parent actor *a* does. Therefore, the generalization_generation refactoring for actors is a behavior preserving transformation. The refactoring is shown in Figure 4.17.

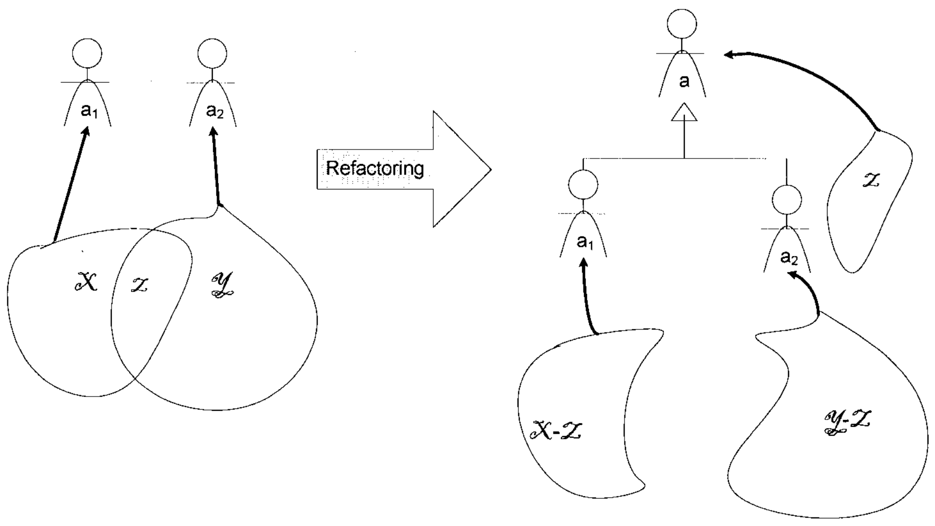


Figure 4.17 Generalization_generation refactoring for actors

4.3.7 Precedence_generation refactoring

Motivation

As we known, precedence relationship is a special case of inclusion relationship. A precedence relationship can be factored out from an inclusion relationship when the inclusion point is positioned at the very beginning of the episode tree of the base use case.

In other words, the included use case is always invoked in the very beginning when the base use case is executed. In such situation, a precedence relationship instead of an inclusion relationship is suggested to expose chronological order between use cases clearly.

The specific process is: a precedence relationship from the included use case to the base use case is generated; the original inclusion relationship is deleted; the pseudo episode in the base use case, that indicates the inclusion point, is deleted either. The refactoring helps to discover the chronological relationship between use cases in a clearer way.

Definition

Preconditions:

- The base use case *u* with its episode tree *t*;
- The included use case *u'*, with its episode tree t_i;
- The first child of *t* is a pseudo episode t_i', which indicates the inclusion point for the inclusion relationship from the base use case *u* to the included use case *u'*;
- The inclusion relationship *r* from the base use case *u* to the included use case *u'*, where the inclusion point in the base use case is t_i'.

Parameters:

- *u*: the base use case;
- *t*: the episode tree of use case *u*;
- *u'*: the included use case;
- t_i': the pseudo episode in *t*, indicates the inclusion point;
- *r*: the inclusion relationship from use case *u* to use case *u'*.

Postconditions:

- A precedence relationship from use case *u'* to use case *u* is generated;
- The inclusion relationship *r* is deleted from the use case model;
- The pseudo episode t_i' is deleted from the episode tree of the base use case *u*.

Verification

Let us analyze the semantics of the precedence relationship. Use case *u'* precedes use case *u* means that *u'* always takes place prior to *u* when *u* is executed.

From the view of the episode tree: the use case *u* includes the use case *u'* in the specific point before the refactoring; the inclusion point is defined by the pseudo episode t_i' in *t*. Since t_i' is the first child of *t*, the use case *u'* takes place prior to the other sub-trees of the use case *u*. After the refactoring, the use case *u'* takes place prior to the use case *u* because of the precedence relationship. Since t_i' has been deleted from the episode tree of the use case *u* after the refactoring, the rest of the episode tree of the use case *u* will be executed after *u'*. Therefore, the precedence_generation refactoring is a behavior preserving transformation. The refactoring is shown in Figure 4.18.

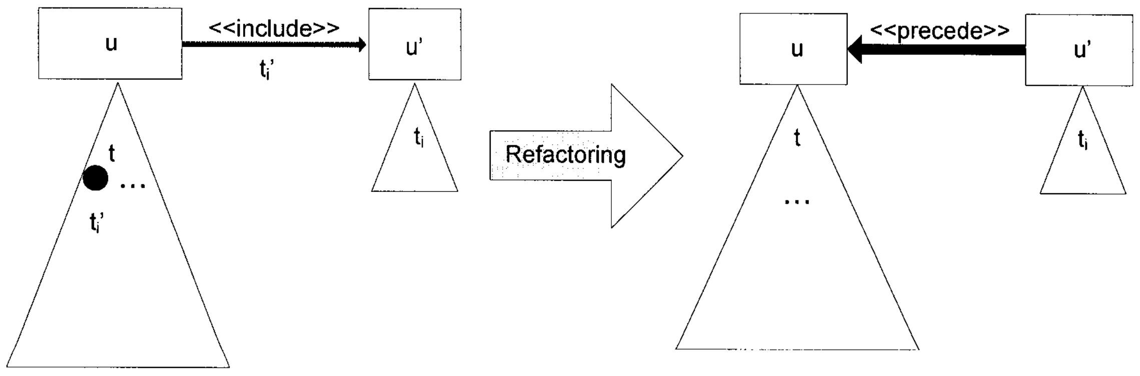
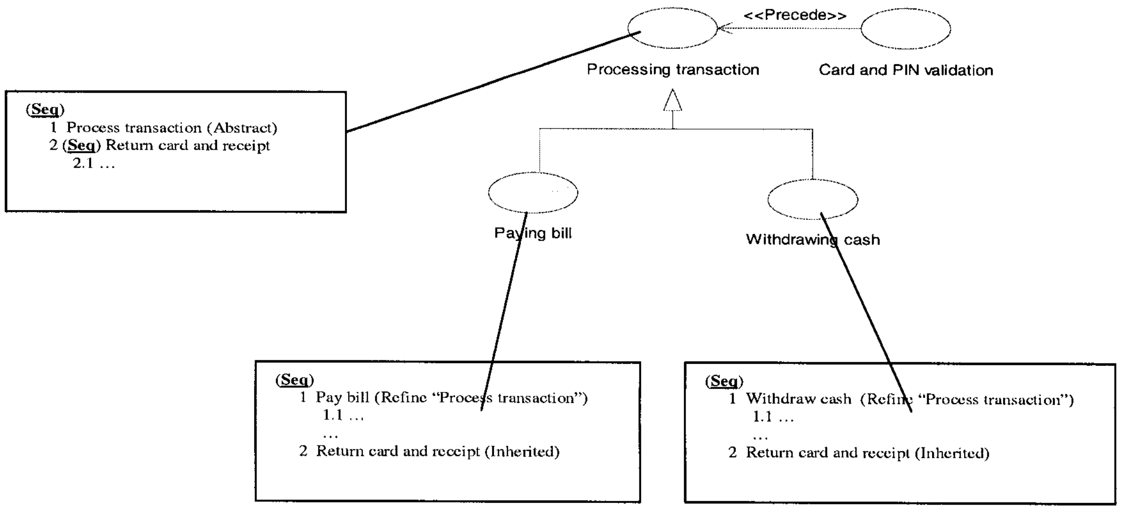


Figure 4.18 Precedence_generation refactoring.

Example

In Figure 4.16, the included use case “Card and PIN validation” always appear at the very beginning of use case “Processing transaction”. To clarify the chronological order between use cases, a precedence relationship is generated. The use case diagram after applying “precedence_generation refactoring” is shown in Figure 4.19.



<<Precede>>
Processing transaction
Card and PIN validation
(Seq)
1 Process transaction (Abstract)
2 (Seq) Return card and receipt
2.1 ...
Paying bill
Withdrawing cash
(Seq)
1 Pay bill (Refine “Process transaction”)
1.1 ...
...
2 Return card and receipt (Inherited)
(Seq)
1 Withdraw cash (Refine “Process transaction”)
1.1 ...
...
2 Return card and receipt (Inherited)

Figure 4.19 Use case diagram after applying “Precedence_generation refactoring”

4.3.8 Inclusion_mergence refactoring

Motivation

If the included use case is too specific or too abstract compared with other use cases, it is better to merge it into the base use case. As a result, the inclusion relationship is removed since the included use case does not exist any more. The episode tree of the included use case is inserted into base use case at the inclusion point. In other words, the pseudo episode in the base use case is substituted by the episode tree of the included use case. If the included use case is redundant after the refactoring, it can be deleted by applying the delete_usecase refactoring rule. The refactoring helps to control use case granularity and maintain similar abstract level among use cases.

Definition

Preconditions:

- The base use case *u* with its episode tree *t*;
- The included use case *u'* with its episode tree t_i;
- the pseudo episode t_i', which indicates the inclusion point;
- The inclusion relationship *r* from *u* to *u'*.

Parameters:

- *u*: the base use case;
- *u'*: the included use case;
- *r*: the inclusion relationship from *u* to *u'*.

Postconditions:

- The pseudo episode t_i' is replaced by the episode tree t_i;
- The relationship *r* is deleted.

Verification

Let us analyze the semantics of the inclusion relationship. The use case *u* includes the use case *u'* means that *u'* is a sub-functionality of *u*. When the use case *u* is executed, the sub-functionality will be invoked at the specific point where t_i' indicates.

Before the refactoring, the pseudo episode t_i' will be replaced by the episode tree t_i of the included use case *u'* when use case *u* is executed because of the inclusion relationship. After the refactoring, the pseudo episode t_i' is substituted by the episode tree t_i; the inclusion relationship is also deleted. As we see, the behavior of the use case has no change after the refactoring. Therefore, the inclusion_mergence refactoring is a behavior preserving transformation. The refactoring is shown in Figure 4.20.

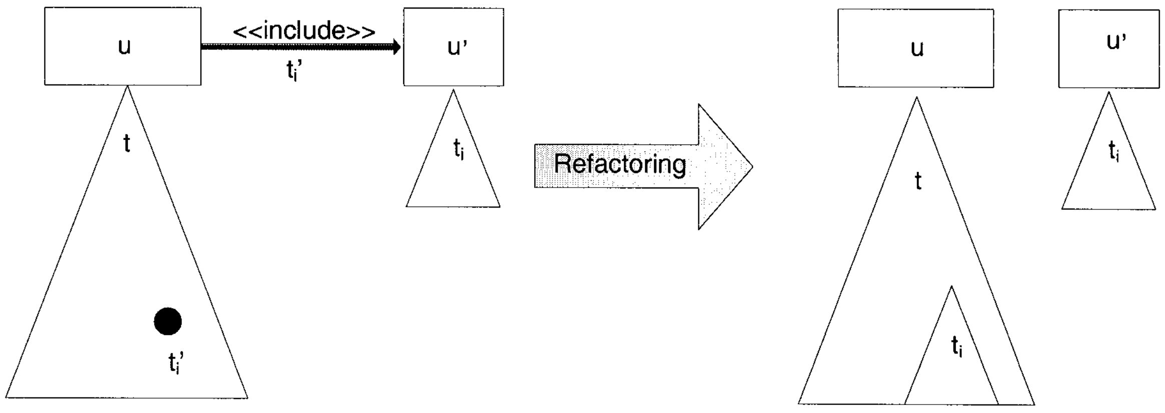


Figure 4.20 Inclusion_mergence refactoring.

4.3.9 Extension_mergence refactoring

Motivation

If the extending use case is too specific or too abstract compared with other use cases, it is better to merge it into the base use case. As a result, the inclusion relationship is removed. The episode tree of the extending use case is inserted into the base use case at the extension point. In other words, the pseudo episode in the base use case is substituted by the episode of the extending use case. If the extending use case is redundant after the refactoring, it can be deleted by applying the delete_usecase refactoring rule. The refactoring helps to control use case granularity and maintain similar abstract level among use cases.

Definition

Preconditions:

- The base use case *u* with its episode tree *t*;
- The extension point t_i', which is a pseudo episode in *t*;
- The extending use case *u'* with its episode tree t_i;
- The extension relationship from *u'* to *u*;

Parameters:

- *u*: the base use case;
- *u'*: the extending use case;
- *r*: the extension relationship from u' to u;

Postconditions:

- The pseudo episode t_i' is substituted by t_i;
- The relationship *r* is deleted;

Verification

Let us analyze the semantics of the extension relationship. The use case *u'* extends the use case *u* means that *u'* will take place when the extension condition is satisfied. The episode tree t_i of *u'* will be inserted into the episode tree *t* of the base use case *u* at the extension point t_i'.

Before the refactoring, the episode tree t_i of the extending use case is inserted into *t* to replace the pseudo episode t_i' if the extension condition is satisfied. After the refactoring, the pseudo episode t_i' is replaced by the episode tree t_i of the extending use case *u'* whether or not the condition is satisfied. However, t_i will never be reached if the condition is not satisfied in the use case *u*. As we see, the behavior of the use case *u* has no change after the refactoring. Therefore, the extension_mergence refactoring is a behavior preserving transformation. The refactoring is shown in Figure 4.21.

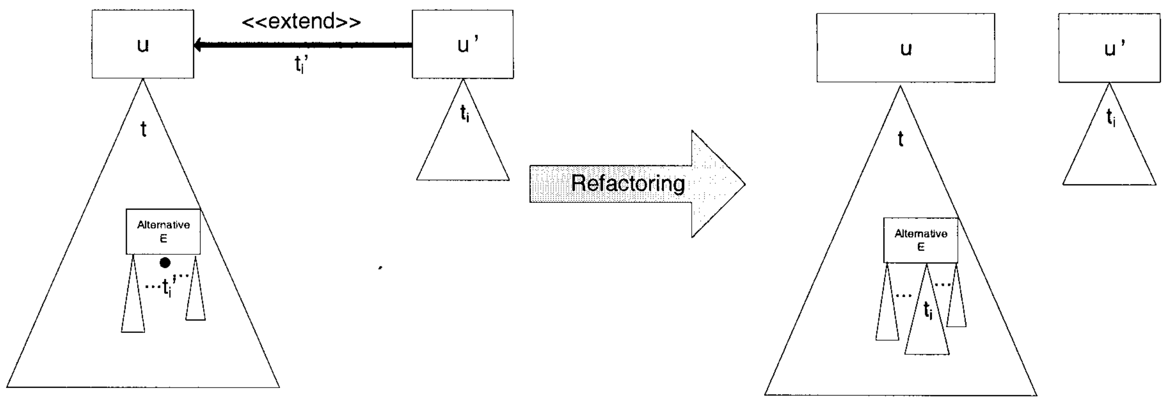


Figure 4.21 Extension_mergence refactoring

4.3.10 Precedence_mergence refactoring

Motivation

When the preceding use case is too specific or too abstract compared with other use cases, it is better to merge it into the base use case. As a result, the precedence relationship is removed. If the preceding use case is redundant after the refactoring, it can be deleted by applying the delete_usecase refactoring rule. The refactoring helps to control use case granularity and maintain similar abstract level among use cases.

Definition

Preconditions:

- The base use case *u* with is episode tree *t*;
- The precedent use case *u'* with its episode tree t_i;
- The precedence relationship from *u'* to *u*.

Parameters:

- *u*: the base use case;
- *u'*: the preceding use case;

- *r*: the precedence relationship from *u'* to *u*;

Postconditions:

- *u'* is merged into *u*: the episode tree t_i is inserted into the episode tree *t* as the first child of the root.
- The relationship *r* is deleted;

Verification

Let us analyze the semantics of the precedence relationship. The use case *u'* precedes the use case *u* means that *u'* always be invoked firstly when *u* is executed.

Before the refactoring, the use case *u'* takes place prior to the base use case *u* because of the precedence relationship from *u'* to u. After the refactoring, the episode tree t_i of the use case *u'* becomes the first sub-tree of *t* of the use case *u*. As we know, the first child will always take place at the very beginning provided that the root of *t* is a sequence composite episode. Therefore, the precedence_mergence refactoring is a behavior preserving transformation. The refactoring is shown in Figure 4.22.

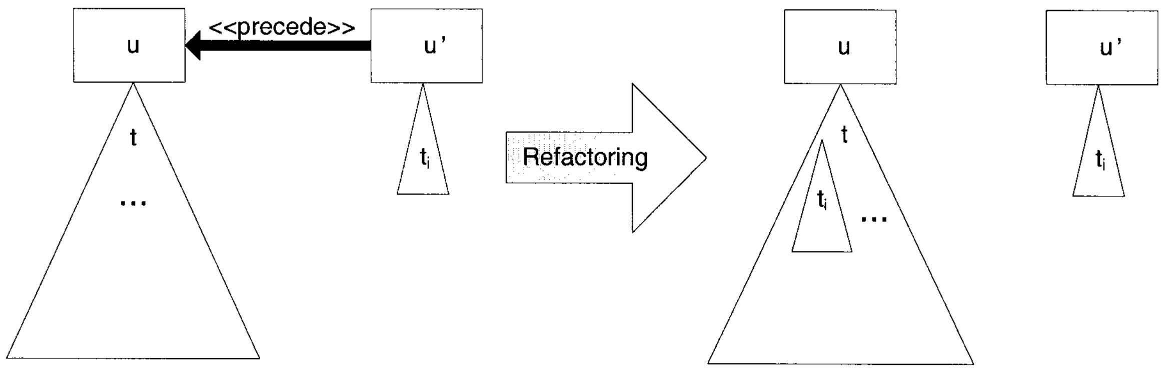


Figure 4.22 Precedence_mergence refactoring

Note: Here we suppose the root of t is a sequence composite episode, otherwise more work will be needed: construct a new sequence composite episode as the root; take t_i as its first child, and *t* as its second child.

4.4 Other refactoring rules

In addition to the refactoring rules introduced in section 4.3, other refactoring rules, such as “reuse_episode”, “delete_episode”, “reuse_event”, and “delete_event”, are based on the information captured in the event models. Also, many basic refactoring rules, such as “Create_UseCase”, “Create_Episode”, “Change_UseCaseName”, “Change_EpisodeName” and so on, are fundamental bricks for composite refactoring rules. We will not introduce these refactoring rules in this thesis.

Chapter 5

A case study

This chapter introduces a case study of the VSS (Video Store System). We start from the system analysis to get an original draft use case model. Afterward, we focus on use case refactoring. The step-by-step transformation of the use case model is demonstrated by applying the use case refactoring rules introduced in chapter 4.

5.1 Introduction

5.1.1 System description

The VSS is a computer-based business management system. The purpose of the VSS is to facilitate staff management of the rentals of movies, video games and related equipments for small chain of 10 stores scattered throughout the city. The system will be able to manage the inventory of rental items, keep track of all rental transactions, rental history and status of each customer member.

The system should provide the following functions [Comp647]:

- **Inventory Management**: Video items will be added to inventory when purchase and receive videotapes, games or equipments. The inventory is maintained by or upon the authorization of the store manager. Each store maintains its own inventory. The inventory is searchable but not updateable by all the other stores.
- **Customer Maintenance**: Customers details are maintained by or upon the authorization of the store manager. Customers will receive customer membership

cards and a password for website access once customers satisfy the prerequisite. The system also issue child dependent membership. Dependent members must be associated with a primary member. All the stores share a common list of customer members so that the customer membership, once established in one store, is valid in all stores.

- **Rental Service**: Customers may rent videotapes, video games, and equipments through a store staff. The system will verify the customer membership, check condition, and record transaction details. Various rental conditions must be applied to a child dependent member what and when the child can rent.
- **Reservation Service**: Customer may reserve videotapes, video games, and equipments for a specific date. Staff will receive an on-screen reminder of such reservations when a returned item is available.
- **Inquiry Service**: Staff will be able to search for movie titles on behalf of the customer. The search will result in location identification. If the item not found in the store, it is possible to search the other stores.
- **Web Service:** Once Login in the website of the video store using card and password, the customer can search and reserve an item including videotape, game, or equipment through Internet.

The above services VSS provide may be modified and expanded given the intended clients desire for such modification.

5.1.2 Actor characteristics

The primary actors of the VSS are the video store staff. However, the customers of the video store will interact directly with the web service module of the system via the

Internet. Therefore, actors in this system include [Comp647]:

- **Manager Staff**: Manager Employee or owner of the video store. Tasks performed by these staff members cover all of the functionality provided by the VSS, namely inventory management, customer maintenance, video rentals, video reservation, video return, video inquiries, and promotion service.
- **Staff:** An employee, who is the primary actor of the system. Tasks performed by these staff members cover the majority of the functionality provided by the VSS, including video rentals, video reservation, video return, and video inquiries.
- **Customer:** Anyone of the general public who is registered as a member of the video store system. The customer interacts with the system directly only when web-reservation or web-search is a task. The Staff member or Manager Staff actor performs all the other services of the VSS on behalf of the customer.
- **Dependent customer:** Any child under 18 year-old of the general public who is registered as a dependent member of a customer. The dependent customer interacts with the system directly only when the web-reservation or web-search is a task. The existence of dependent customer depends on the corresponding primary customer. They use same account for payment, the rental items is accumulated within customer and his/her dependent customers. The Staff member or Manager Staff actor performs all the other services of the VSS on behalf of the dependent customer.

5.2 Use case and episode models

5.2.1 The use case diagram for VSS

According to the above requirement description, a use case diagram is given in Figure 5.1.

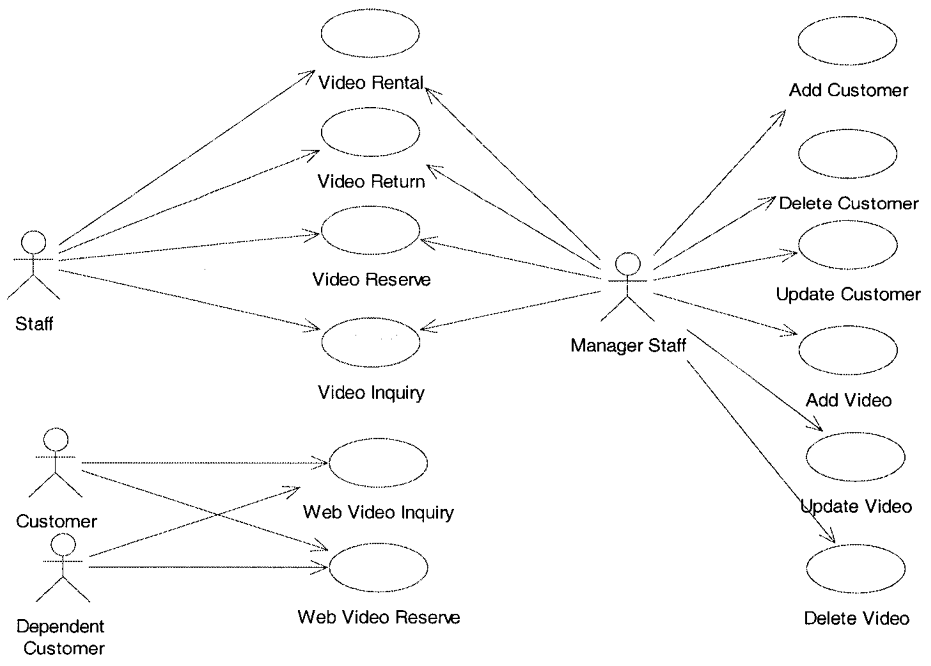


Figure 5.1 Use case diagram for the VSS

5.2.2 Episode models

According the 3-level metamodel, we analyze each use case in the use case diagram and generate its episode model to discover the use case's internal structure of behavior. The episode tree for each use case is shown as follows.

5.2.2.1 Video Rental

The episode tree for "Video Rental" is shown as follows:

```
(Seq)
  1 The customer is identified
  2 The available video item is identified by the system.
  3 (Alt) The system checks if satisfying the condition
    3.1 Yes: A new video loan is registered.
    3.2 No: Return message.
```

5.2.2.2 Video Return

The episode tree for “Video Return” is shown as follows:

```
(Seq)
  1 The customer is identified.
  2 The borrowed video item is identified
  3 The loan is identified through the item
  4 The corresponding loan is removed
  5 Calculate the rental fee.
  6 (Alt) If there is a reservation for this item
    6.1 Yes: return reminder message
    6.2 No: continue
  7 Print out the rental fee.
```

5.2.2.3 Video Reserve

The episode tree for “Video reserve” is shown as follows:

```
(Seq)
  1 The customer is identified
  2 A video title is identified
  3 (Alt) Check if satisfying the condition
    3.1 Yes: A reservation is created with the specified title and customer
    3.2 No: Return error message
```

5.2.2.4 Video Inquiry

The episode tree for use case “Video inquiry” is shown as follows:

```
(Seq)
  1 The user inputs the search criteria
  2 The system executes database inquiry
  3 Return search result
```

5.2.2.5 Add Customer

The episode tree for the use case “Add Customer” is shown as follows:

```
(Seq)
  1 Input the necessary information for the customer
  2 Database inquiry
  3 (Alt) if the customer already exist in database?
    3.1 Yes: Return message
    3.2 (Seq) No: Issue the card for the customer
      3.2.1 Add the customer with corresponding constraints to database
      3.2.2 Generate card to customer
      3.2.3 Return succeed message
```

5.2.2.6 Update Customer

The episode tree for the use case "Update Customer" is shown as follows:

```
(Seq)
  1 The customer is identified
  2 Database inquiry
  3 (Alt) if the customer already exist in database?
    3.1 (Seq) Yes: Update customer's information
      3.1.1 Input the necessary information for the customer
      3.1.2 Update the database
      3.1.3 Return succeed message
    3.2 No: Return error message
```

5.2.2.7 Delete Customer

The episode tree for the use case "Delete Customer" is shown as follows:

```
(Seq)
  1 The customer is identified
  2 Database inquiry
  3 (Alt) if the customer already exist in database?
    3.1 (Seq) Yes: Delete the customer
      3.1.1 Delete the customer from the database
      3.1.2 Return succeed message
    3.2 No: Return error message
```

5.2.2.8 Add Video

The episode tree for the use case "Add Video" is shown as follows:

```
(Seq)
  1 Input the necessary information for the video
  2 Database inquiry
  3 (Alt) if the video item exists already
    3.1 Yes: Return message that the video already exists
    3.2 (Seq) No: Add the video item
      3.2.1 Add video item to database
      3.2.2 Return succeed message
```

5.2.2.9 Update Video

The episode tree for the use case “Update Video” is shown as follows:

```
(Seq)
  1 The video item is identified
  2 Database inquiry
  3 (Alt) if the video item exists already
    3.1 (Seq) Yes: Update the video’s information
      3.1.1 Input the information for the video item
      3.1.2 Update the database
      3.1.3 Return succeed message
        3.2 No: Return error message
```

5.2.2.10 Delete Video

The episode tree for the use case “Delete Video” is shown as follows:

```
(Seq)
  1 The video item is identified
  2 Database inquiry
  3 (Alt) if the video item exists already
    3.1 (Seq) Yes: Delete the video item
      3.1.1 Delete the item from the database
      3.1.2 Return succeed message
    3.2 No: Return error message
```

5.2.2.11 Web Video Inquiry

The episode tree for the use case “Web Video Inquiry” is shown as follows.

```
(Seq)
  1 The customer login in the website
  2 Input search criteria
  3 The system executes database inquiry
  4 Return search result
```

5.2.2.12 Web Video Reserve

The episode tree for the use case “Web Video Reserve” is shown as follows.

```
(Seq)
  1 The customer login in the website
  2 A video title is identified
  3 (Alt) Check if satisfying the condition
    3.1 Yes: A reservation is created with the specified title and customer.
    3.2 No: Return error message
```

5.3 Use case refactorings

Step 1 Generalization_Generation refactoring for actors:

From the use case model, we find that actors "Staff" and "Manager Staff" share common relationships to several use cases. Furthermore, these actors belong to similar family. According to the definition of generalization_generation refactoring for actors, we can apply the "generalization_generation refactoring for actors" to these two actors. Similarly, we can also apply the "generalization_generation refactoring for actors" to actors "Customer" and "Dependent Customer". The refactored use case model is shown in Figure 5.2.

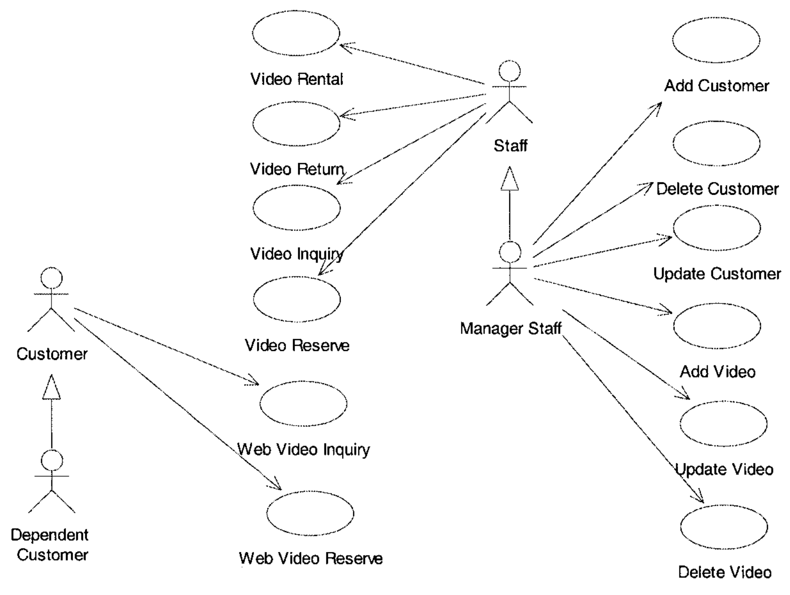


Figure 5.2 Use case diagram after step 1

The actor "Dependent Customer" inherits the association relationships to "Web Video Inquiry" and "Web Video Reserve" from its parent actor "Customer" after the refactoring. Similarly, the actor "Manager Staff" inherits the association relationships to

"Video Rental", "Video Return", "Video Inquiry", and "Video Reserve" from its parent actor "Staff" after the refactoring.

Step 2 Inclusion_generation refactoring:

From the episode tree of the use case "Web Video Inquiry", we notice that part of the episode tree describes a relatively independent functionality. Therefore we can extract this functionality as a sub-tree of the episode tree, shown as follows:

```
(Seq)
  1 The customer login in the website
  2 (Seq) Video search
    2.1 Input search criteria
    2.2 The system executes database inquiry
    2.3 Return search result
```

According to the "inclusion_generation refactoring", we can break up the use case into two smaller use cases, named "Web Video Inquiry" and "Video Search". The use case diagram after the refactoring is shown in Figure 5.3.

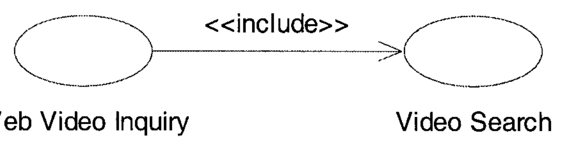


Figure 5.3 Use case diagram for "Web Video Inquiry" after step 2

An inclusion relationship from "Web Video Inquiry" to "Video Search" is generated. According to the episode model, there must be at least one inclusion point defined in the base use case "Web Video Inquiry". The episode tree for use case "Web Video Inquiry" is shown as follows.

```
(Seq)
  1 The customer login in the website
  2 Video search (Inclusion)
```

Additionally, the sub-tree is removed from the base use case "Web Video

Inquiry" and becomes the episode tree of the including use case "Video Search". The episode tree for the new use case "Video Search" is shown as follows.

```
(Seq)
  1 Input search criteria
  2 The system executes database inquiry
  3 Return search result
```

Step 3 Equivalent_generation refactoring:

From the episode trees of use cases "Video Search" and "Video Inquiry", we notice that the use cases have equivalent episode trees. Therefore, the "equivalent_generation refactoring" can be applied. The use case diagram after the refactoring is shown is Figure 5.4.

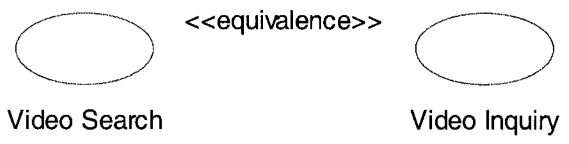


Figure 5.4 Use case diagram after step 3

Because the refactoring does not change the behavior of the use cases, the episode trees for the use cases have no change after the refactoring.

Step 4 Reuse_use case refactoring:

Since use cases "Video Inquiry" and "Video Search" are equivalent, we can also apply "reuse_usecase refactoring" to reduce redundancy and improve the reusability of the system. The use case diagram after applying the refactoring is shown in Figure 5.5.

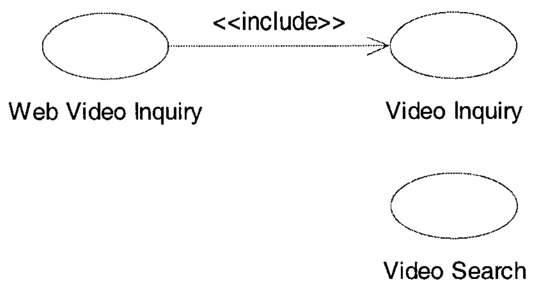


Figure 5.5 Use case diagram after applying "reuse_usecase refactoring"

The relationships (the inclusion relationship in this case) with "Video Search" are taken over by "Video Inquiry". Therefore, the inclusion point that is related to this inclusion relationship has to be changed too. The inclusion point defined in the episode tree of the base use case "Web Video Inquiry" is changed from "Video Search" to "Video Inquiry", as shown in the following episode tree of the use case "Web Video Inquiry".

```
(Seq)
   1 The customer login in the website
   2 Video Inquiry (Inclusion)
```

Step 5 Delete_usecase refactoring:

After applying "reuse_usecase refactoring", the use case "Video Search" has no relationship to any other use cases or actors. Therefore, it can be deleted from the system by applying "delete_usecase refactoring".

Step 6 Generalization_generation refactoring for use cases:

From the episode trees of use cases "Add Customer", "Delete Customer", and "Update Customer", we notice that these use cases are isomorphic. They have similar behavior structure and share some common primitive episode such as "Database inquiry". Therefore, the generalization_generation refactoring can be applied to these use cases to improve reusability. The use case diagram after the refactoring is shown in Figure 5.6.

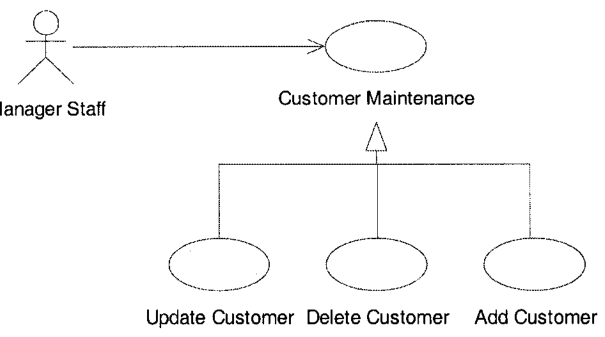


Figure 5.6 Use case diagram for "Customer Maintenance"

After applying "Generalization_generation refactoring", a new parent use case named "Customer Maintenance" is generated. Its episode tree is defined as follows.

```
(Seq)
  1 The system gethers the customer information
  2 Database inquiry
  3 Maintain the customer information
```

As shown in Figure 5.6, the parent use case also takes oven the common relationship with the actor "Manager Staff". The child use case "Add Customer" inherits behaviors from its parent use case "Customer Maintenance" and specifies some of them. Meanwhile, it also inherits relationship with the actor "Manager Staff" from its parent use case. The episode tree for the use case "Add Customer" is changed as follows.

```
(Seq)
  1 Input the necessary information for the customer (Inherited and specified)
  2 Database inquiry (Inherited)
  3 Maintain customer information(Inherited and specified)
    3.1 (Alt) if the customer already exist in database?
      3.1.1 Yes: Return message
      3.1.2 (Seq) No: Issue the card for the customer
        3.1.2.1 Add the customer with corresponding constraints to database
        3.1.2.2 Generate card to customer
        3.1.2.3 Return succeed message
```

Likewise, the use case "Update Customer" not only inherits the relationship with the actor "Manager Staff" from its parent use case "Customer Maintenance", but also inherits behaviors and specifies some of them. Its episode tree is shown as follows.

```
(Seq)
  1 The customer is identified  (Inherited and specified)
  2 Database inquiry (Inherited)
  3 Maintain customer(Inherited and specified)
    3.1 (Alt) if the customer already exist in database?
      3.1.1 (Seq) Yes: Update customer's information
        3.1.1.1 Input the necessary information for the customer
        3.1.1.2 Update the database
        3.1.1.3 Return succeed message
      3.1.2 No: Return error message
```

In the same way, the episode tree of another child use case "Delete Customer" is changed as follows.

```
(Seq)
  1 The customer is identified (Inherited and specified)
  2 Database inquiry (Inherited)
  3 Maintain customer(Inherited and specified)
    3.1 (Alt) if the customer already exist in database?
      3.1.1 (Seq) Yes: Delete the customer
        3.1.1.1 Delete the customer from the database
        3.1.1.2 Return succeed message
      3.1.2 No: Return error message
```

Similarly, use cases "Add Video", "Delete Video", and "Update Video" are isomorphic. They have similar behavior structure and share some common primitive episode such as "Database inquiry". Therefore, the generalization_generation refactoring can be applied to these use cases to improve reusability. The use case diagram after applying the refactoring is shown in Figure 5.7.

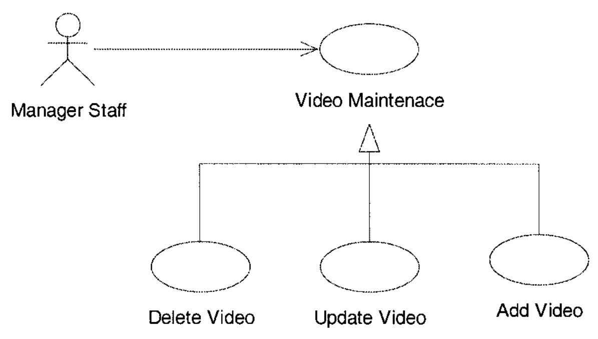


Figure 5.7 Use case diagram for "Video Maintenance"

After applying "Generalization_generation refactoring", the episode tree for the new parent use case "Video Maintenance" is generated as follows. The parent use case also takes over the common relationship with the actor "Manager Staff" from its children use cases.

```
(Seq)
  1 The system gathers the video information
  2 Database inquiry
  3 Maintain the video information
```

After the refactoring, the child use case "Add Video" inherits not only the relationship but also the behavior from its parent use case. The episode tree for the use case is shown as follows.

```
(Seq)
  1 The staff inputs the necessary information for the video (Inherited and specified)
  2 Database inquiry (Inherited)
  3 Maintain video (Inherited and specified)
    3.1 (Alt) if the video item exists already
      3.1.1 Yes: Return message that the video already exists
      3.1.2 (Seq) No: Add the video item
        3.1.2.1 Add video item to database
        3.1.2.2 Return succeed message
```

In the same way, the episode tree for another child use case "Update Video" is modified as follows.

```
(Seq)
  1 The video item is identified Input (Inherited and specified)
  2 Database inquiry (Inherited)
  3 Maintain the video information(Inherited and specified)
    3.1 (Alt) if the video item exists already
      3.1.1 (Seq) Yes: Update the video's information
        3.1.1.1 Input the information for the video item
        3.1.1.2 Update the database
        3.1.1.3 Return succeed message
      3.1.2 No: Return error message
```

Also, the episode tree for another use case "Delete Video" is changed as follows.

(**Seq**)
1 The video item is identified (*Inherited and specified*)
2 Database inquiry (*Inherited*)
3 Maintain the video information (*Inherited and specified*)
  3.1 (**Alt**) if the video item exists already
    3.1.1 (**Seq**) *Yes*: Delete the video item
      3.1.1.1 Delete the item from the database
      3.1.1.2 Return succeed message
    3.1.2 *No*: Return error message

The Final Use Case Diagram After Refactorings

After the above refactorings, the structure of the use case model is changed, shown in Figure 5.8.

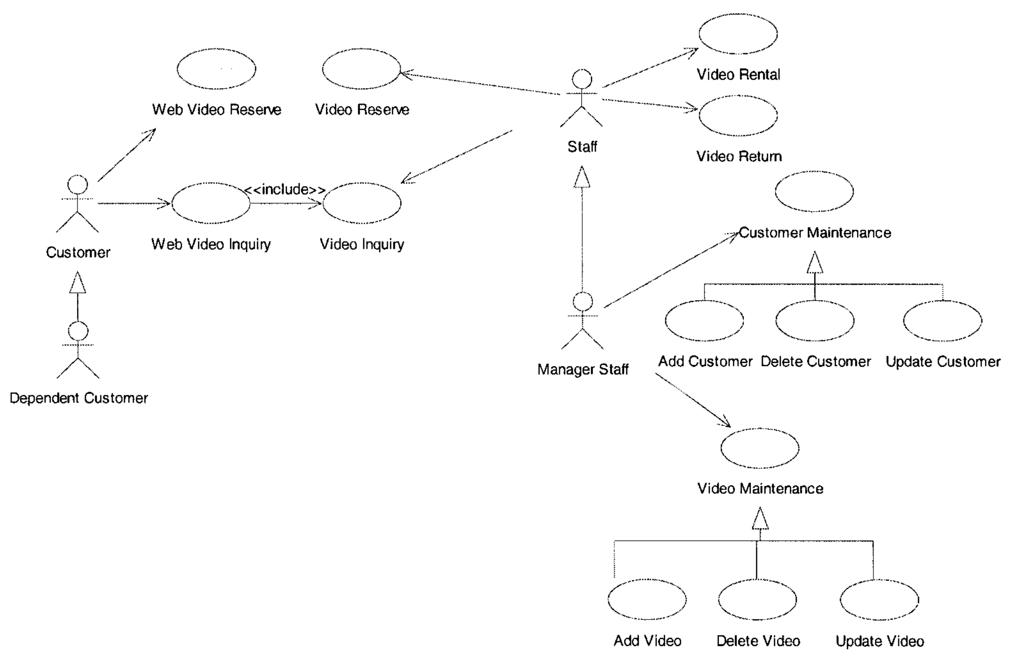


Figure 5.8 Use case diagram for the VSS after applying refactoring rules

The episode tree for the use case "Video Rental" is shown as follows:

(**Seq**)
1 The customer is identified
2 The available video item is identified
3 (**Alt**) Check if satisfying the condition
  3.1 *Yes*: A new video loan is registered.
  3.2 *No*: Return message.

The episode tree for the use case “Video Return” is shown as follows:

```
(Seq)
  1 The customer is identified.
  2 The borrowed video item is identified
  3 The loan is identified through the item
  4 The corresponding loan is removed
  5 Calculate the rental fee.
  6 (Alt) If there is a reservation for this item
    6.1 Yes: return reminder message
    6.2 No: continue
  7 Print out the rental fee.
```

The episode tree for the use case “Video Reserve” is shown as follows:

```
(Seq)
  1 The customer is identified
  2 A video title is identified
  3 (Alt) Check if satisfying the condition
    3.1 Yes: A reservation is created with the specified title and customer
    3.2 No: Return error message
```

The episode tree for the use case “Video Inquiry” is shown as follows:

```
(Seq)
  1 The customer or staff inputs the search criteria
  2 The system executes database inquiry
  3 Return search result
```

The episode tree for the use case “Web Video Reserve” is shown as follows:

```
(Seq)
  1 The customer logins in the website
  2 A video title is identified
  3 (Alt) Check if satisfying the condition
    3.1 Yes: A reservation is created with the specified title and customer.
    3.2 No: Return error message
```

The episode tree for the use case “Web Video Inquiry” is shown as follows:

```
(Seq)
  1 The customer logins in the website
  2 Video Inquiry (Inclusion)
```

The episode tree for the use case “Customer Maintenance” is shown as follows:

```
(Seq)
  1 The system gathers the customer information
  2 Database inquiry
  3 Maintain the customer information
```

The episode tree for the use case "Add Customer" is shown as follows:

```
(Seq)
  1 Input the necessary information for the customer (Inherited and specified)
  2 Database inquiry (Inherited)
  3 Maintain the customer information(Inherited and specified)
    3.1 (Alt) if the customer already exist in database?
      3.1.1 Yes: Return message
      3.1.2 (Seq) No: Issue the card for the customer
        3.1.2.1 Add the customer with corresponding constraints to database
        3.1.2.2 Generate card to customer
        3.1.2.3 Return succeed message
```

The episode tree for the use case "Delete Customer" is shown as follows:

```
(Seq)
  1 The customer is identified (Inherited and specified)
  2 Database inquiry (Inherited)
  3 Maintain the customer information(Inherited and specified)
    3.1 (Alt) if the customer already exist in database?
        3.1.1 (Seq) Yes: Delete the customer
          3.1.1.1 Delete the customer from the database
          3.1.1.2 Return succeed message
        3.1.2 No: Return error message
```

The episode tree for the use case "Update Customer" is shown as follows:

```
(Seq)
  1 The customer is identified  (Inherited and specified)
  2 Database inquiry (Inherited)
  3 Maintain the customer information(Inherited and specified)
    3.1 (Alt) if the customer already exist in database?
      3.1.1 (Seq) Yes: Update customer's information
        3.1.1.1 Input the necessary information for the customer
        3.1.1.2 Update the database
        3.1.1.3 Return succeed message
      3.1.2 No: Return error message
```

The episode tree for the use case "Video Maintenance" is shown as follows:

```
(Seq)
  1 The system gathers the video information
  2 Database inquiry
  3 Maintain the video information
```

The episode tree for the use case "Add Video" is shown as follows:

```
(Seq)
  1 Input the necessary information for the video (Inherited and specified)
  2 Database inquiry (Inherited)
  3 Maintain the video information (Inherited and specified)
    3.1 (Alt) if the video item exists already
      3.1.1 Yes: Return message that the video already exists
      3.1.2 (Seq) No: Add the video item
        3.1.2.1 Add video item to database
        3.1.2.2 Return succeed message
```

The episode tree for the use case "Delete Video" is shown as follows:

```
(Seq)
  1 The video item is identified (Inherited and specified)
  2 Database inquiry (Inherited)
  3 Maintain the video information (Inherited and specified)
    3.1 (Alt) if the video item exists already
      3.1.1 (Seq) Yes: Delete the video item
        3.1.1.1 Delete the item from the database
        3.1.1.2 Return succeed message
      3.1.2 No: Return error message
```

The episode tree for the use case "Update Video" is shown as follows:

```
(Seq)
  1 The video item is identified (Inherited and specified)
  2 Database inquiry (Inherited)
  3 Maintain the video information (Inherited and specified)
    3.1 (Alt) if the video item exists already
      3.1.1 (Seq) Yes: Update the video's information
        3.1.1.1 Input the information for the video item
        3.1.1.2 Update the database
        3.1.1.3 Return succeed message
      3.1.2 No: Return error message
```

In this case study, we applied 6 refactoring rules, which are: generalization_generation refactoring for actors, inclusion_generation refactoring, equivalence_generation refactoring, reuse_usecase refacotoring, delete_usecase refactoring, and generalization_genertion refactoring for use cases. After these refactorings, the organization of the use case model is improved. With the tool's support, the refactoring will be more efficient.

Chapter 6

Conclusion

Use case models are used to capture the functional requirements of a system. However, to obtain well-organized use case models is not easy. This thesis shows how refactoring as a concept is applied in use case modeling to improve the organization of use case models. The work the author has done, the limitations and the future work for use case refactoring are presented.

6.1 Achievements of this thesis

Based on the analysis of Regnell and Rui's use case metamodel and Metz's refined use case model, the author refined the use case metamodel at the environment level, namely the use case model, by introducing more relationships and detailing the inclusion and extension points. Constraints applied to the model are given in OCL. Also, I refined the use case metamodel at the structure level, namely the episode model, by classifying different episodes and introducing control-flow to composite episodes. Similarly, some constraints applied to the episode model are given in OCL. The semantics of the episode model are explained in informal text.

To apply refactoring to use case modeling, the author defined more than 10 use case refactoring rules. Most of these rules change use case models based on the information captured in episode models. Each rule is explained in detail, including motivation, definition, and verification. An example based on the ATM is given following the rule to demonstrate how the rule is applied in practice. These rules focus on how to restructure the composition of use cases, and rearrange the relationships between use cases.

A case study of the Video Store System (VSS) is given to demonstrate step-by-step how the refactoring rules provided in this thesis are applied to change the organization of use case models.

6.2 Limitations

Although we refined the metamodel at the environment and structure levels and presented the constraints among objects using OCL, the semantics of the model is informal. This makes it difficult to describe the behaviors of a use case precisely. For our project, we use a glossary to reduce inconsistency.

For some refactoring rules, such as generalization_generation refactoring for use cases, the preconditions and/or postconditions are hard to verify. In this thesis, the problem is simplified and only part of these conditions is illustrated.

We focused on the environment and structure levels when refining the metamodel. The refactoring rules and the case study are also limited to these two levels.

6.3 Future work

To precisely depict the behavior of a use case, a formal language is needed to define the semantics of the use case model, the episode model, and the event model. Future work includes choose a suitable formal language and use it to describe use cases, episodes, and events.

Refactoring in use case modeling is still in its infancy. We introduced some refactoring rules. These rules focused on the decomposition of a use case and the reorganization of relationships between use cases. More refactoring rules that influence the organization of use case models are expected with future research on use case refactoring. Furthermore, future work may include the refactoring rules' standardization and formalization to make the rules more accurate and unambiguous.

In our tool, we only implemented the refactoring rules that are provided by Rui. Future work is to implement the refactoring rules introduced in this thesis. Moreover, a user may want to go back to the previous models after he or she applied several refactoring rules. Therefore, future work for the tool includes providing redo and undo operations for users.

We introduced case studies based on the Video Store System (VSS) and the Automated Teller Machine (ATM). However, a small numbers of case studies do not fully demonstrate the effectiveness of the refactoring in use case modeling. More case studies are needed.

References

[Armour01] Frank Armour, Granville Miller, *Advanced use case modeling*, Addison - Wesley. 2001

[Booch99] G. Booch, I. Jacobson, and J. Rumbaugh, *Unified Modeling Language – Users Guide*. Addison Wesley Longman, Inc. Reading, MA, 1999.

[Butler01] Greg Butler, Lugang Xu, *Cascading Refactoring for Framework Evolution, Proceedings of 2001 Symposium on Software Reuseablity, p51-57, ACM Press*, 2001.

[Comp647] Eric Atisso, Gang Cheng, Shao Zhen Fang, Gabriel Schor, Fan Wang, Ju Wang, Tie Bang Wang, Wei Yu, Bing Zhu, *Project Report of the Video Store System for Course Comp647: Software Methodologies*, Concordia University, Team2, 2001.

[Fowler99] Martin Fowler, *Refactoring: Improving the Design of Existing Code,* Addison-Wesley, 1999.

[Jacobson92] Jacobson I., Christerson M., Josson P., Overgarrd G., *Object-oriented Software Engineering: A Use Case Driven Approach.* Addison Wesley. 1992.

[Larman98] Craig. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design.* Prentice Hall. Upper Saddle River, NJ, 1998.

[Mens04] Tom Mens, Martin Fowler, Chris Seguin, Don Roberts, Ralf Laemmel, Simon Thompson, Claus Reinke, Serge Demeyer, Dirk Janssens, *Program Refactoring*, http://www.program-transformation.org/Transform/ProgramRefactoring, 2004.

[Metz01a] Pierre Metz, John O'Brien, Wolfgang Weber. *Use case model refactoring: Changes to UML's use case relationships*, http://www.fbi.fhdarmstadt.de/frames/organisation/personen/w.weber/public_html/Publ/use_case_model_refacturing-internal_report.pdf, 2001.

[Metz01b] Pierre Metz, John O Brien, Wolfgang Weber. *Against use case interleaving*, http://www.wibas.de/download/Against_Use_Case_Interleaving.pdf, 2001.

[Miller01] Roy W. Miller, Christopher T. Collins, *XP distilled*, IBM developerWorks, March 2001.

[Opdyke92] W.F.Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992.

[Opdyke00] Bill Opdyke, *Refactoring, Reuse & Reality*, Lucent Technologies/ Bell Labs, 2000.

[Regnell96] B. Regnell, M. Andersson, and J. Bergstand. *A hierarchical use case model with graphical representation*. Proceedings of ECBS'96, P270, IEEE International Symposium and Workshop on Engineering of Computer-Based Systems, 1996. Also available at http://www.tts.lth.se/Personal/bjornr/Papers/ECBS96.pdf.

[Regnell99] B. Regnell, *Requirements Engineering with Use Cases – a Basis for Software Development*. Ph.D. thesis, Lund University, 1999.

[Roberts99] Donald Bradley Roberts. *Practical Analysis for Refactoring*. Ph.D. thesis, University of Illinois, 1999.

[Rui01] KeXing Rui, *Refactoring Use Case Models, Thesis Proposal*, Concordia University, Computer Science, 2001.

[Scott02] Kendall Scott, *UML Explained*, Addison-Wesley, p39, 2002.

[Simons99] Anthony J H Simons, *Use Cases Considered Harmful*, http://www.dcs.shef.ac.uk/~ajhs/papers/harmful.pdf, 1999.

[Tokuda99] Lance Tokuda. *Evolving Object-Oriented Designs with Refactorings.* Ph.D. thesis, University of Texas, 1999.

[UML01] *OMG Unified Modeling Language Specification 1.4*, http://www.ift.ulaval.ca/~bui/21454_Hiv2003/OMG_14.pdf, 2001

[Wiki03] Wiki web pages, *Refactor*, http://c2.com/cgi/wiki?ReFactor, 2003.

[Wiki04] Wiki web pages, *What Is Refactoring*? 2004. http://c2.com/cgi/wiki?WhatIsRefactoring, 2004.