Refactoring Use Case Models: The Metamodel

Kexing Rui **Greg Butler**

Department of Computer Science
Condordia University
Montréal H3G 1M8, Canada
Email: {rui,gregb}@cs.concordia.ca

Abstract

This paper describes how refactoring as a concept can be broadened to apply to use case models. A metamodel for use case modeling is described in detail, which represents our perspective on use case formalization. This metamodel allows us to define several categories of use case refactorings that help us discover and organize use case refactorings. A list of current refactorings is given. Finally, we illustrate the concept of use case refactorings with a simple example.

Keywords: refactoring, use case, use case model, metamodel, task, goal, episode, software maintenance

1 Introduction

Software maintenance is the most difficult part in software production. During the software life cycle, more time and cost are attributed to maintenance than to any other phase. In fact, about two-thirds of total software costs is devoted to maintenance (Schach 1999).

Software maintenance activities require changes to the software system. Usually, software changes can be viewed from different levels. At the low level, it is in terms of lines of code to be changed. Software restructuring is most often used at this level to make the source code easier to maintain. At the high level, it is in terms of features to be added to a system. Use Cases are a means of describing the functionality of a system. During software maintenance, it is a challenge to update use case models consistently.

We extend the concept of refactoring from source code to use case models. This paper describes how refactoring as a concept can be broadened to apply to use case models.

A refactoring is a change made to the internal structure of a software component to make it easier to understand and cheaper to modify, without changing the observable behavior of that software component (Fowler 1999). Therefore, refactoring is a behavior-preserving program transformation. Some transformations can be automated to update an application's design and underlying source code. Since the introduction the term "*refactoring*" in early 1990s (Opdyke 1992), refactoring has been an active research area. It is gaining more recognition because of its central role in eXtreme Programming (XP).

John McGregor and his colleagues (Miller et al. 1999) developed use case assortment for the requirements analysis phase of framework development. Use case assortment organizes the use case model by introducing abstract actors and use cases, and by rearranging responsibilities. This work inspired cascaded refactoring (Butler & Xu 2001), which introduced use case refactoring, as well as architectural refactoring. This paper is a deeper study of use case refactoring.

In the following sections, we introduce some related work on refactoring, use case modeling, and cascaded refactoring. Then we present our use case metamodel, which represents our perspective on use case formalization. This metamodel is the foundation for our use case refactorings. We also present several categories of use case refactorings and a list of current refactorings. Finally, a simple example illustrates the concepts of use case refactoring.

Copyright ©2003, Australian Computer Society, Inc. This paper appeared at the Twenty-Fifth Australasian Computer Science Conference (ACSC2003), Adelaide, Australia. Conferences in Research and Practice in Information Technology, Vol. 16. Michael Oudshoorn, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

2 Related Work

Our research on refactoring use case models is closely tied to the following work.

2.1 Refactoring

William F.Opdyke's Ph.D. thesis (Opdyke 1992) is the first publication on refactoring. He identifies seven invariants required to preserve the behavior of a C++ program and he defines 26 low level refactorings, which are listed in Table 1. He also defines three high level refactorings, which are more abstract. Because these small refactorings are correct under certain conditions, large changes that are composed solely of small refactorings are also correct.

Based on Opdyke's research, Lance Tokuda (Tokuda 1999) goes further to evolve object-oriented designs with refactorings. His research shows that all three kinds of design evolution, which are schema transformations, design pattern microarchitectures, and the hot-spot-driven-approach, are automatable with refactorings.

To make refactoring more practical, Donald Bradley Roberts (Roberts 1999) develops a commercial-grade tool, the Refactoring Browser for Smalltalk. There are now several refactoring tools for Java programs, for example from OTI (see `www.eclipse.org`). Martin Fowler (Fowler 1999) provides guidelines for the refactoring process and explains the principles and best practices of refactorings.

Refactoring is gaining more recognition through eXtreme Programming (XP) (Beck 1999). In XP people usually start with a simple design and allow it to evolve. One of the key aspects of XP is continual refactoring of the source code.

2.2 Use Case Modeling

Ivar Jacobson (Jacobson et al. 1992) is regarded as the inventor of use cases. He defines them as follows: "*a use case is a specific way of using the system by using some part of the functionality. A use case constitutes a complete course of interaction that takes place between an actor and the system.* "

Since then, use cases have been widely used in requirement gathering and domain analysis. With the release of the Unified Modeling Language (UML) specification version 1.1, the scope of use cases has broadened to include modeling constructs at all levels. Currently, there are various approaches to describe and formalize use cases, which represent different perspectives on use case modeling.

Jacobson, Griss, and Jonsson (Jacobson et al. 1994) propose requirement gathering through use scenarios, first captured informally, then expressed more formally in a use case model. They introduce responsibility into use case description so that use cases can provide usage-oriented view of component system documentation.

Cockburn (Cockburn 1997) identifies four dimensions to use case descriptions: purpose, content, plurality, and structure. He introduces a theory based on a small model of communication, distinguishing "goals" as a key element of use cases.

Regnell (Regnell 1999) investigates the role of use case modeling in requirements engineering and its relation to system verification and validation. His approach allows a hierarchical structure and enables graphical representation at different abstraction levels. He defines use case syntax and semantics.

Buhr (Buhr 1998) has developed Use Case Maps (UCMs) as a visual notation for comprehending and developing the architecture for emergent behavior in large, complex, self modifying systems. UCMs consist of three primary constructs: responsibilities, causality, and components. UCMs can be refined through decomposition and partitioned by factoring.

UML (Rumbaugh et al. 1999) represents the merger of three main contributing methodologies: OMT, Booch, and Objectory. However, UML's definition for use case has shifted from Jacobson's original emphasis on use to a more system-centric viewpoint. According to UML: a use case is the specification of sequences of actions, including variant sequences and error sequences, that a system, subsystem, or class can perform by interacting with outside actors.

The OPEN Modeling Language (OML) is a competing meta-modeling to the UML. It represents the merger of SOMA (Henderson-Sellers et al. 1997), MOSES (Henderson-Sellers & Firesmith 1997), and Firesmith (Firesmith et al. 1997). A key principle behind OML is the notion of tasks and techniques. In OML, capturing user requirements involves the use of task scripts, which are supported by a task-action grammar. These task scripts can be organized into composition, classification and usage structures.

Creating a program entity
create_empty_class
create_member_variable
create_member_function
Deleting a program entity
delete_unreferenced_class
delete_unreferenced_variable
delete_member_functions
Changing a program entity
change_class_name
change_variable_name
change_member_function_name
change_type
change_access_control_mode
add_function_argument
delete_function_argument
reorder_function_arguments
add_function_body
delete_function_body
convert_instance_variable_to_pointer
convert_variable_references_to_function_calls
replace_statement_list_with_function_call
inline_function_call
change_superclass
Moving a member variable
move_member_variable_to_superclasses
move_member_variable_to_subclasses
Intermediate level (composite) refactorings
abstract_access_to_member_variable
convert_code_segment_to_function
move_class

Table 1: A List of Refactorings by Opdyke

2.3 Cascaded Refactoring

Cascaded refactoring (Butler & Xu 2001) is a hybrid approach for the development and evolution of frameworks. It combines the modeling aspects of top-down domain engineering approaches and the iterative, refactoring approaches of the bottom-up object-oriented community. It weaves together steps for partial domain engineering to better understand the domain and how to evolve the current working set of partial models, and steps of system refactoring and extension. It stresses traceability between models and defines an alignment of models to be a traceability mapping that is consistent with the mapping of common and variable features. The set of models that are used in cascaded refactoring are a feature model, a use case model, an architectural model, a design and the source code.

Cascaded refactoring relates the set of refactorings across the set of models through change impact analysis using the trace maps. The impact of the refactorings for one model is translated via the trace maps to determine constraints on the refactorings of another model.

Cascaded refactoring is still in its infancy. Although the concept of refactoring use case models is proposed in cascaded refactoring, and a few example refactorings given, the details of use case refactorings are investigated in this paper here.

3 Refactoring Use Case Models

3.1 Use Case Metamodel

There has been a strong debate about the use and semantics of use cases. Different people are using use cases differently. Use cases are the medium to model user requirements for requirement engineering and they are used to guide the design of communicating objects to satisfy functional requirements for software engineers. While in the hands of user interface designers, they become a task model for understanding user needs and guiding user interface design. Although UML represents some kind of effort in use

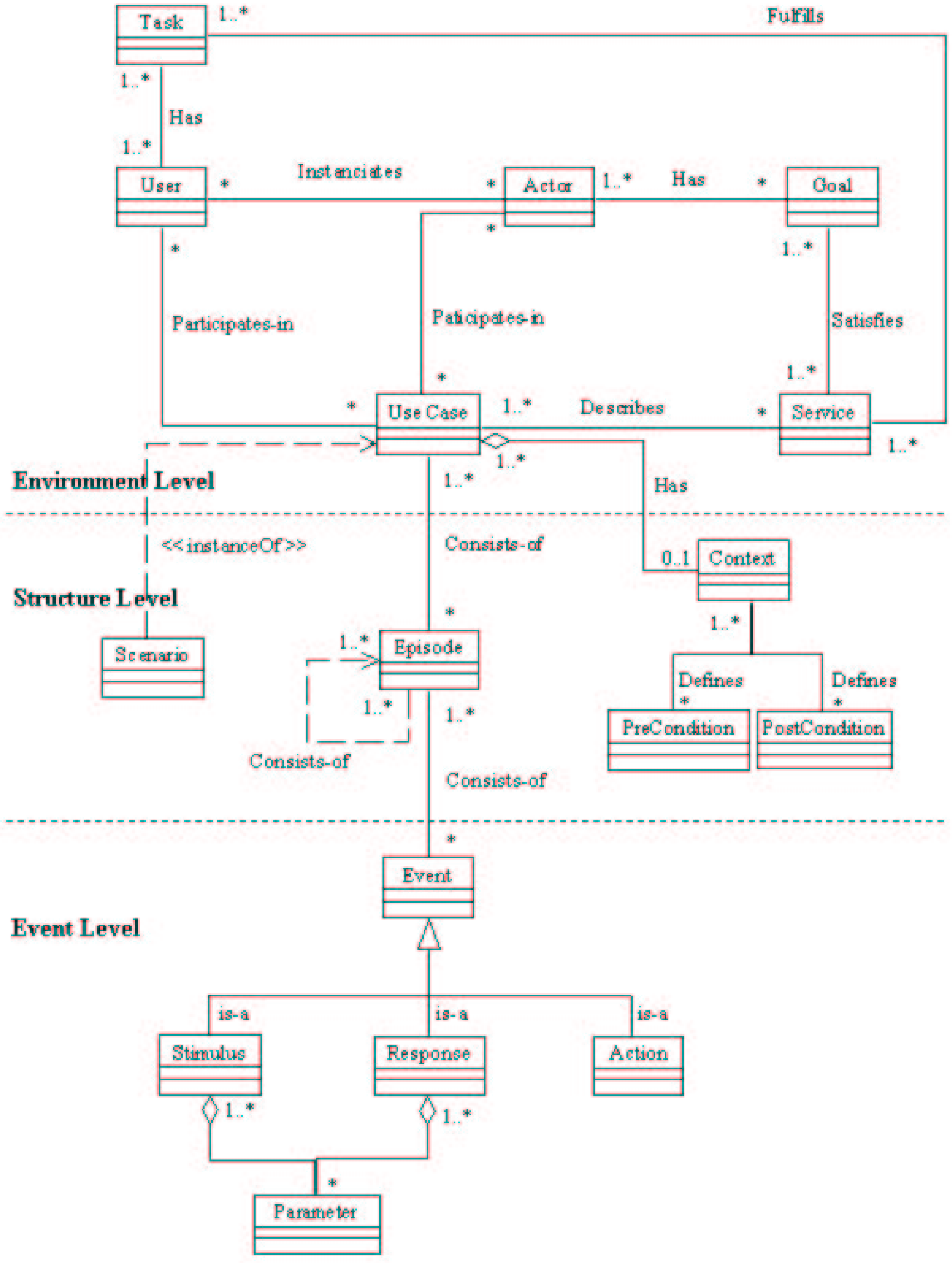

Figure 1: Use Case Metamodel

case formalization, there is still a lot to argue about this issue.

As the first step for refactoring use case models, we have to define use case semantics clearly. We do not intend to define a formal use case semantics. Our focus is to provide good support for requirements engineering, software engineering and user interface design. Therefore, we define the use case metamodel as shown in Figure 1, which represents our perspective on use case formalization.

This metamodel incorporates Regnell's use case specification (Regnell 1999) as well as other people's work in use case modeling. It can be viewed from different levels. At the environment level, the use case is related to the entities external to intended system. At the structure level, the internal structure of a use case is revealed together with its different variants and parts. The event level represents a lower abstraction level where the individual events are characterized.

In our metamodel, a *use case* is a system usage scenario characteristic of a specific actor. A use case models a usage situation where one or more *services* of the target system are used by one or more *users* with the aim to accomplish one or more *goals*. A use case may either model a successful or an unsuccessful accomplishment of goals. An *actor* is a specific role played by a system user, and represents a category of users that demonstrate similar behavior when using the system. A user is regarded as an instance of an actor. The user can be human beings, and other external systems or devices communicating with the system. One user may appear as several instances of different actors depending on the context. *Goals* are objectives that users have when using the services of a target system. Goals are used to categorize users into actors. *Task* describes what the user will do with the software system. 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 *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. A use case may be divided into coherent parts, called *episodes*. The same episode can occur in many use cases. A use case may have a *context*, which demarcates the scope of the use case and defines its *pre-conditions* and *post-conditions*. *Precondition* is the property of the environment and the target system that need to be fulfilled in order to invoke the use case. *Postcondition* is the property of the environment and the target system at use case termination.

An *event* is the specification of a significant occurrence that has a location in time and space. It can be a *stimulus*, *response* or *action*. *Stimulus* is the message from users to the target system. *Response* is the message from the target system to users. *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. Stimulus and response can take *parameters*, which carry data to and from the target system.

We define different kind of relationships as shown in Table 2. 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 incorporates 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 which 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.

Use Case Relationship
inclusion
extension
generalization
precedence
similarity
equivalence
Actor Relationship
generalization
Task Relationship
inclusion
generalization
Goal Relationship
inclusion
generalization

Table 2: A Summary of Relationships

An *extension* relationship between two 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.

A *generalization* relationship between two use cases implies that the child use case contains all the attributes, sequences of behavior, 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.

A *similarity* relationship between two use cases defines 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.

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

A *precedence* relationship between two use cases defines that one use case is sequenced (appended) to the behavior of the preceding use case.

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 par-

	Objectory Jacobson	SOMA Graham	OML Firesmith	UML Rational	Our Metamodel
Use Case Relationship	uses extends	usage composition specialization	invokes precedes	includes extends generalization	inclusion extension generalization precedence similarity equivalence

Table 3: Terminology for Use Case Relations

ent actor, i.e. communicate with the same set of use cases, as the parent actor.

An *inclusion* relationship between two tasks defines that one task contains another task, which is a subtask. One subtask may be included in several other tasks and one task may include several other subtasks. 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).

An *inclusion* relationship between two goals defines that one goal may contain another goal, which is a subgoal. One subgoal may be included in several other goals and one goal may include several subgoals. 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).

3.2 Metamodel Justification

Since our metamodel is not a formal semantics, there cannot be a theoretical proof or analysis of its completeness, consistency and other properties. What we want to do here is to compare with what other people have done with use case modeling and to check whether our metamodel covers the range of uses of use cases in requirements engineering, software engineering and user interface design.

Our metamodel is based on Regnell's use case model (Regnell 1999), which can be a basis for requirements engineering. We introduce the concept of task to provide better support for user interface design. Task describes what the user will do with the software system. We define task relationship and goal relationship so that people can organize a hierarchical task structure or goal structure.

In task analysis, the basic technique is the decomposition of a whole into its parts. The task decomposition is a structured way of developing information from a task description into a series of more detailed declarations about particular issues. A hierarchical task model is created by breaking task into increasingly detailed task elements. The most general information is placed at the top of the hierarchy, while the more specific information following on lower levels. In this hierarchical task model, the relationship between task and subtask can be described by inclusion and generalization, which are defined in our metamodel.

Goal-driven approaches focus on why systems are constructed, expressing the rationale and justification for the proposed system. Organizing goals hierarchically provides a good way to represent relationships between goals and subgoals (Anton et al. 1994, Anton 1997, Dardenne et al. 1993). Cockburn suggests to distinguish "goals" as a key element of use cases and structure use cases with goals (Cockburn 1997). Nielsen describes a technique (Nielsen 1994) for extending a task analysis based on the goal composition, which is suggested by Clayton Lewis (Lewis 1990). Basically, goal composition starts by considering each primary goal that the user may have when using the system. A list of possible additional features is then generated by combining each of these goals with a set of general meta-goals that extend the primary goals. In these approaches, the relationship between goal and subgoal can be expressed by inclusion and generalization as defined in our metamodel.

In our metamodel, we define a comprehensive set of use case relations. As shown in Table 3, which compares the terminologies, our metmodel covers that of use case relations as described for Objectory (Jacobson et al. 1992), for SOMA (Graham 1994), in the OPEN Modeling Language (OML) reference manual (Firesmith et al. 1997) and the Unified Modeling Language (UML) version 1.3 (Jacobson et al. 1999) as well as those in our approach. Among these relations, uses-relation is semantically equivalent with an includes-relation. In fact, uses-relation is defined in UML version 1.1(Rational 1997), but it is replaced by includes-relation in UML version 1.3. Basically, usage-relation, composition-relation and invokes-relation can be expressed by includes-relation. Since both specialization-relation and generalization-relation aim at a hierarchical structure with inheritance, we only keep generalization-relation in our metamodel. Besides these relations, we add similarity-relation because it provides a way to carry forward insight about relationships among use cases when the exact nature of the relationship is not yet clear. We add equivalence-relation because it can be very useful when a single definition covers two or more different intentions from the user's perspective. As Larry Constantine (Constantine & Lockwood 2001) says: "*it makes it easier to validate the model with users and customers while also assuring that only one design will be developed.* "

In our metamodel, the event level describes the detailed interaction between the system and actors, which is very important for design, implementation and testing. Therefore, it can be very useful to software engineers.

4 Primitive Use Case Refactorings

Our research indicates that we can extend the notion of refactoring that has been applied to source code and transform use case models with use case refactorings. We categorize use case refactorings as follows. A list of current refactorings are listed in Table 4.

4.1 Creating a Use Case Entity

This category includes refactorings regarding creating new use case, actor, user, task, goal, service, episode, and so on.

4.2 Deleting a Use Case Entity

This category includes refactorings regarding deleting unreferenced use case, actor, user, task, goal, service, episode, and so on.

4.3 Changing a Use Case Entity

This category includes refactorings regarding changing the name of use case, actor, user, task, goal, service, episode, and so on. It also includes refactorings about changing inheritance relationship such as `change_parent_usecase`, `change_parent_actor`, `change_parent_task`, `change_parent_goal`. The refactoring `change_parent_usecase`, for example, is to establish an inheritance relationship between two use cases.

4.4 Moving an Element of Use Case

This category includes refactorings such as `move_actor_to_parent_usecase`, `move_goal_to_parent_actor`, `move_episode_to_parent_usecase`, and so on.

4.5 Distributing Behavior

This category includes refactorings such as `decompose_usecase`, `decompose_goal`, `decompose_task`, and so on. The refactoring `decompose_usecase`, for example, is to split one use case into two use cases.

5 An Example of Use Case Refactoring

This section presents a small example of the application of use case refactorings. The example system we will use is the classic Automated Teller Machine (ATM) that is used in many textbooks. To simplify the illustration, we only list two use cases as shown in Figure 2. The use case Withdrawing Cash offer the service for customers to get cash from ATM. The use case Checking Balance is for customers to check their account balance. Figure 3 presents a structural view for use case Withdrawing Cash and Checking Balance. Both use cases have the episode of Card and PIN Validation.

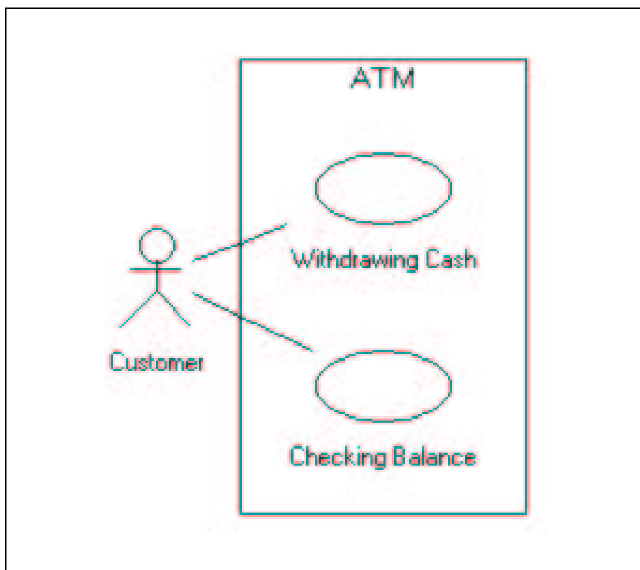

Figure 2: Use Case Model for ATM

```
Creating a use case entity
create_empty_usecase
create_empty_actor
create_empty_user
create_empty_task
create_empty_goal
create_empty_service
create_empty_context
create_empty_precondition
create_empty_postcondition
create_empty_episode
create_empty_event

Deleting a use case entity
delete_unreferenced_usecase
delete_unreferenced_actor
delete_unreferenced_user
delete_unreferenced_task
delete_unreferenced_goal
delete_unreferenced_service
delete_empty_context
delete_empty_precondition
delete_empty_postcondition
delete_unreferenced_episode
delete_unreferenced_event

Changing a use case entity
change_usecase_name
change_actor_name
change_user_name
change_task_name
change_goal_name
change_service_name
change_precondition_name
change_postcondition_name
change_episode_name
change_event_name
change_parent_usecase
change_parent_actor
change_parent_task
change_parent_goal

Moving an element of use case
move_actor_to_parent_usecase
move_actor_to_child_usecase
move_goal_to_parent_actor
move_goal_to_child_actor
move_service_to_parent_usecase
move_service_to_child_usecase
move_episode_to_parent_usecase
move_episode_to_child_usecase

Distributing behavior
decompose_usecase
decompose_goal
decompose_task
```

Table 4: Current Use Case Refactorings

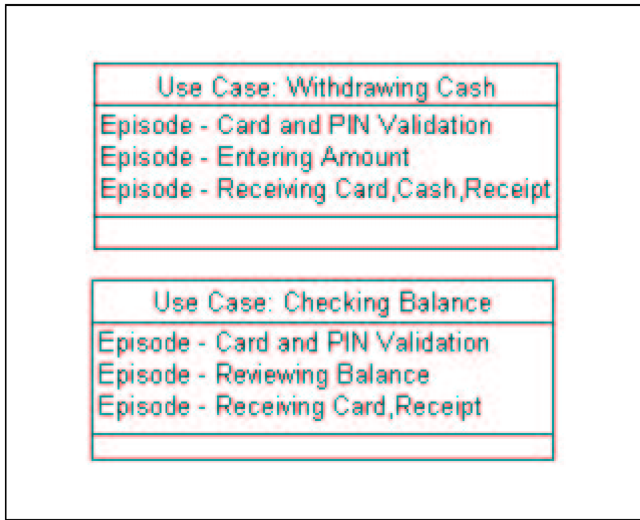

Figure 3: Structure View

In the maintenance scenario, our ATM is to be enhanced to allow transfers between accounts. It is noted that this use case also requires an episode to validate the user's card and PIN, so the developer will first restructure the use case model to promote the reuse of this episode or use case, and then proceed to extend the system with the new functionality for transfers between accounts.

We can transform the use case model (as shown in Figure 2 and Figure 3) into one promoting reuse by applying use case refactorings in three steps.

Step 1. Create a new use case Customer Transaction using `create_empty_usecase` refactoring.

Step 2. Make the use case Customer Transaction to be the parent use case of Withdrawing Cash and Checking Balance using `change_parent_usecase` refactoring (Figure 4).

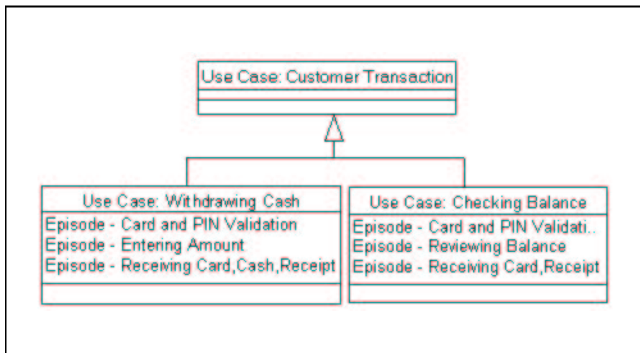

Figure 4: After Step 2: apply `change_parent_usecase`

Step 3. Move up the episode Card and PIN Validation using `move_episode_to_parent_usecase` (Figure 5). As a result, the new use case model for ATM is shown in Figure 6.

6 Conclusion

Our research demonstrates that we can extend the concept of refactoring from source code to use case models. We define use case semantics informally through our metamodel. Based on this metamodel, we describe several refactoring categories that we are using to discover and organize use case refactorings. A list of current refactorings is presented. A simple example illustrates the application of use case refactorings.

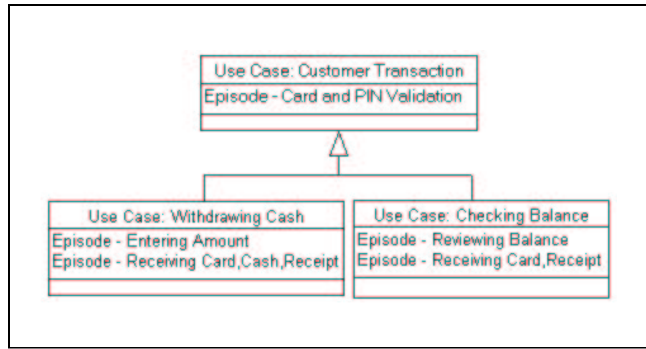

Figure 5: After Step 3: apply `move_episode_to_parent_usecase`

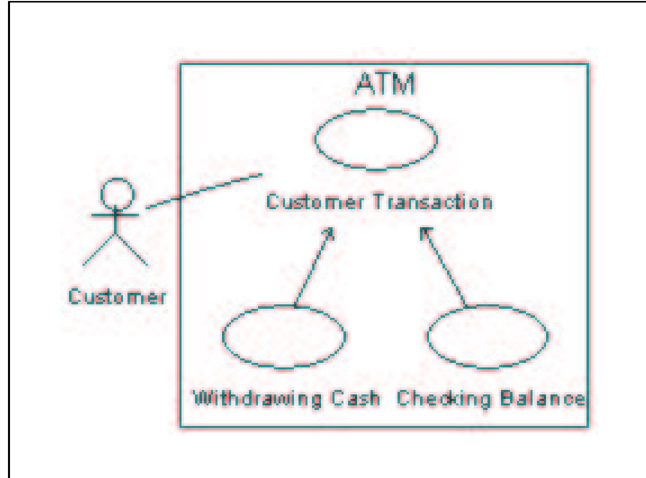

Figure 6: New Use Case Model for ATM

In future work we hope to demonstrate that use case refactoring can be used to improve the maintainability of use case models. We are working on several larger case studies chosen from the standard software engineering literature, such as the ATM (automated teller machine), university library borrowing system, the simpler video store borrowing system, and the elevator control simulator. For these we are developing use case models covering the three levels — environmental, structural, and event — and developing "scenarios" of how the systems might evolve.

As the next step, we will identify what notion of "behavior" is appropriate for use case refactorings when viewed as behaviour-preserving transformations. We will validate that each refactoring is behavior preserving. Furthermore, we will continue to discover and define more use case refactorings to facilitate use case evolution.

Since performing these refactorings manually can be time consuming and error prone, we are developing a tool for use case modeling and refactoring. This tool will provide us with a way to validate our use case refactorings by applying them to the maintenance of larger scale software systems. Developing such a tool will also give us insight into the criteria for a successful refactoring process, and how to best provide tool support for the process. We are going to put our use case metamodel into the repository of this tool. The tool will support zoom-in and zoom-out to explore use case models. Overviews and detailed views will be presented at the environment level, structure level and event level respectively.

References

Anton, A. I. (1997), *Goal identification and refinement in the specification of software-based information systems*, Ph.D.thesis, Georgia Institute of Technology.

Anton, A. I. & McCracken, W. M. & Potts, C. (1994), *Goal decomposition and scenario analysis in business process reengineering, advanced information systems engineering*, 6th International Conference Proceedings (CaiSE'94), pp.94–104.

Beck, K. (1999), *Extreme Programming Explained: Embrace Change*, Addison-Wesley.

Buhr, R. J. A. (1998), *Use Case Maps as Architectural Entities for Complex Systems.* Transactions on Software Engineering, IEEE, Vol. 24, No. 12, pp. 1131-1155.

Butler, G. & Xu, L. (2001), *Cascaded refactoring for framework evolution*, Proceedings of 2001 Symposium on Software Reusability, ACM Press, pp. 51–57.

Cockburn, A. (1997), *Structuring use cases with goals*, JOOP/ROAD 10(5) and 10(7).

Constantine, L. L. & Lockwood, L. A. D. (2001), *Structure and style in use cases for user interface design*, In Mark Van Harmelen, editor, *Object Modeling and User Interface Design*, Addison-Wesley.

Dardenne, A. & Van Lamsweerde, A. & Fickas, S. (1993), *Goal-directed requirements acquisition*, Science of Computer Programming, 20, 3–50.

Firesmith, D. G. & Henderson-Sellers, B. & Graham, I. (1997), *OPEN Modeling Language (OML) Reference Manual*, SIGS, New York.

Fowler, M. (1999), *Refactoring: Improving the Design of Existing Code*, Addison-Wesley.

Graham, I. (1994), *Migrating to Object Technology*, Addison-Wesley.

Henderson-Sellers, B. & Firesmith, D.G. (1997), *Choosing between UML and OPEN*, American Programmer, 10(3), 15-23.

Henderson-Sellers, B. & Firesmith, D. G. & Graham, I. (1997), *OML Metamodel: Relationships and State Modeling*, JOOP 10, 1, March/April.

Jacobson, I. & Booch, G. & Jonsson, P. & Overgaard, G. (1992), *Object-Oriented Software Engineering: A Use-Case Driven Approach*, Addison-Wesley.

Jacobson, I. & Booch, G. & Rumbaugh, J. (1999), *The Unified Software Development Process*, Addison Wesley.

Jacobson, I. & Ericsson, M. & Jacobson, A. (1994), *The Object Advantage: Business Process Reengineering with Object Technology*, ACM Press.

Lewis, C. (1990), *A research agenda for the nineties in human-computer interaction*, Human-Computer Interaction, 5, 125–143.

Miller, G. G. & McGregor, J.D. & Major, M. L. (1999), *Capture framework requirements*, In M. E. Fayad, D. C. Schmidt, and R. E. Johnson, editors, *Building Application Frameworks: Object-Oriented Foundations of Framework Design.* Wiley.

Nielsen, J. (1994), *Goal composition: extending task analysis to predict things people may want to do*, http://www.useit.com/papers/goalcomposition.html

Opdyke, W. F. (1992), *Refactoring object-oriented frameworks*, Ph.D. thesis, University of Illinois.

Rational Corporation, (1997), *UML Summary, Semantics, Notation Guide, Version 1.1*, Rational Software Corporation.

Regnell, B. (1999), *Requirements engineering with use cases — a basis for software development*, Ph.D. thesis, Lund University.

Roberts, D. B. (1999), *Practical analysis for refactoring*, Ph.D. thesis, University of Illinois.

Rumbaugh, J. & Jacobson, I. & Booch, G. (1999), *The Unified Modeling Language Reference Manual*, Addison-Wesley.

Schach,, S. R. (1999), *Software Engineering with JAVA*, McGraw-Hill.

Tokuda, L. (1999), *Evolving object-oriented designs with refactorings*, Ph.D. thesis, University of Texas at Austin.