On Refactoring of Use Case Models

Jian Xu

A Thesis

in

The Department

of

Computer Science

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

February 2004

Your file Votre référence
ISBN: 0-612-91141-1
Our file Notre référence
ISBN: 0-612-91141-1

ABSTRACT

On Refactoring of Use Case Models

Jian Xu

Use case models are widely used in software engineering. It is important to improve the understandability and maintainability of use case models. Refactoring is a behavior-preserving transformation. The research shows that refactoring as a concept can be broadened to apply to use case models to improve their understandability, changeability, reusability and traceability. In this thesis a use case metamodel is described for use case modeling in detail. Then some refactoring rules for the use case metamodel are defined and implemented. Based on the Drawlets framework, a prototype tool is implemented for defining the use case models and applying refactorings to the models. A case study is also presented to illustrate the practical use of these refactorings. The experience shows that the tool facilitates the refactoring process greatly.

Acknowledgements

I would like to thank my supervisor, Prof. Gregory Butler, for his guidance, encouragement and inspiration. Without his suggestions and helps, the work would not be what it is today.

I would also like to thank Kexing Rui who shares his great idea with us and helps us on making the idea come true.

I also want to thank the people in our research group, Wei Yu and Renhong Luo, for their helps, contributions and experiences so that I can finish my work as my might expect.

Finally, I wish to dedicate the work to my family, especially to my wife Xiuhui, for their loves, supports and encouragements throughout the entire process of my master program study.

Table of Contents

List of Figures

List of Tables

1. Introduction

Requirements change. It happens sometimes right before the system is going to be released. Those projects to be done in a very short time will take more risks. It is obvious that to make a clear and complete requirement specification is one of the key points of finishing a software system with all the required features on time. Different ways have been tried and applied to building the requirement model as well as refactoring on the model in recent years. Use case model is one of the very popular ways of constructing the requirement model. It shows the functional requirements of the software system. In the use case model, some elements are used to describe the model. For example, a use case represents a series of transactions between the actor and the system. An actor represents a certain user type or a role played by users. It is a challenge to manage use case model effectively during software evolution.

Source code refactoring has already been used in mature development products and enhanced the quality of codes greatly. But to make clearly what the users want and define the requirements as precise and complete as possible is always better than altering the system design and architecture in the later stage or just before the system to be released. I choose to work on extending the concept of refactoring from source code to use case model in order to get better-described requirement model. My work attempts to show

how refactoring as a concept can be broadened to apply to use case models to improve their understandability, changeability, reusability and traceability.

1.1 Objective

In the thesis, a use case metamodel and some refactoring rules are introduced. Based on the extension of Regnell's use case metamodel, a three-level use case metamodel is used to record different aspects of system requirements. A refactoring tool is designed and implemented to help people define the use case model and apply refactoring rules to the use case model easily. The implementation of the metamodel and the refactoring rules in the refactoring tool is also discussed. In the case study, the use case model of an ATM system is created and improved by using refactoring rules according to the changes on requirements. The experiences with the tool show that it greatly facilitates the defining and refactoring process.

The whole project is teamwork. I worked with other two graduate students. We all took part in the following work of the project:

a) discuss with Rui about the use case metamodel and its implementation

b) discuss with Rui about the definition of the refactoring rules and the implementation of the rules in the refactoring tool

c) discuss and design the general system structure and data structure of the refactoring tool

Besides the work done together with other two students, I did the following work of the project independently:

a) understand the concept of use case model including Regnell's use case metamodel and refactoring

b) study and extend Drawlets framework to be the base framework of the tool

c) design data structure and system structure of use case model on environment level

d) implement use case model on environment level in the tool

e) implement refactoring rules related to use case model on environment level

f) use the tool to make an case study which is a use case model of ATM system

The implementation of goal and task model editor and related refactoring rules was done by Renhong Luo. The implementation of episode and event model editor and related refactoring rules was done by Wei Yu.

1.2 Scope of Thesis

The thesis is organized as follows. In section two, requirement engineering and use case model are introduced. In section three, our own use case metamodel is discussed. In section four, refactoring is mentioned and the refactoring rules are defined for the use case metamodel. In section five, Drawlets framework is described. Also the system design, modules and data structure of the tool are described in the same section. In section six, a case study is used to show the way of defining a use case model with the tool. The thesis is summarized and the future work is discussed in section seven.

2. Requirement Engineering and Use Case Model

2.1 Requirement Engineering

The essence of software engineering is constructing and maintaining computer-based systems. This construction typically commences with Requirement Engineering (RE). The major objective of Requirement Engineering is defining the purpose of a proposed system and outlining its external behaviors. RE also gives strong support for system design, implementation and testing.

Requirement Engineering activities can be divided into five categories: [29]

1) Requirements elicitation which is the process of exploring, acquiring, and reifying user requirements through discussion with the problem owners, introspection, observation of the existing system, task analysis and so on.
2) Requirements modeling where alternative models for target composite system are elaborated and a conceptual model of the enterprise as seen by the systems eventual users are produced. This model is meant to capture as much of the semantics of the real world as possible and is used as the foundation for an abstract description of the requirements.

3) Requirements specification where the various components of the models are precisely described and possibly formalized to act as a basis for contractual purposes between the problem owners and the developers.
4) Requirements validation where the specifications are evaluated and analyzed against correctness properties (such as completeness and consistency), and feasibility properties (such as cost and resources needed).
5) Requirements management refers to the set of procedures that assists in maintaining the evolution of requirements throughout the development process. These include planning, traceability, and impact assessment of changing requirements and so on.

A software requirement is a feature, functionality, or property that a software product must have: [26]

1) A user-level capability (e.g., the spreadsheet software should provide a facility for computing the standard deviation)
2) A specific constraint on the software system (e.g., all column and row summations should be updated every 15 seconds)

2.2 Use Case Model

In order to have a well-documented description of the system requirements, use case was introduced and developed in the past years. After years of development, use cases have been a powerful and widely recognized tool for functional requirements elicitation.

Following are some people who made great contributions on use case concept and its usage.

Ivar Jacobson is regarded as the inventor of use cases. He defined use case 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" [15]. Although this definition is brief, broad, and imprecise, his introduction of use cases immediately improved the situation that requirement documents often have a poor fit with both business reengineering and implementation. 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. They represent different perspectives on use case modeling.

Jacobson, Griss, and Jonsson provide a thorough methodology addressing architectural, process, and organizational aspects of software reuse [16]. They propose requirement gathering through use scenarios, first captured informally, then expressed more formally in a use case model. The use case model is considered to be the starting point for the test model. Each execution of a use case described as a scenario will correspond to one test case. They paid considerable attention to component systems and variability mechanisms. Variability in component systems occurs at variation points and utilizes one of three mechanisms: inheritance, configuration, and parameterization.

Inheritance is used to specialize or extend behavior through the <<uses>> and <<extends>> stereotypes. Configuration slots are filled by choosing alternative component implementations. Parameterization can take the form of a bound variable, a template instantiation, or an evaluated expression. Moreover, they introduced responsibility into use case description so that use cases can provide usage-oriented view of component system documentation that enables a developer to learn more quickly how to design from the component systems by seeing how they were intended to be used.

Cockburn identifies four dimensions to use case descriptions: purpose, content, plurality, and structure [5]. Each of these dimensions has an enumerated domain value. Purpose can be either for user stories or requirements. Content can be contradicting, consistent prose, or formal content. Plurality is either 1 or multiple. Structure can be unstructured, semi-formal, or formal structure. He introduces a theory based on a small model of communication, distinguishing goals as a key element of use cases. In so doing, goals and goal failures can be explicitly discussed and tracked. The goals structure is useful for project management, project tracking, staffing, and business process reengineering. His approach is defined as requirements, consistent prose, multiple scenario, and semi formal structure, which is the same as Jacobson s approach.

UCDA (Use case driven analysis) has basic concepts like actors and use cases. 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. By users means both human beings, and other external systems or devices communicating with the system. An actor is regarded as a class and users as instances of this class. One user may appear as several instances of different actors depending on the context. A use case is a system usage scenario characteristic of a specific actor. A number of typical use cases for every actor are identified during the analysis. Use cases are expressed in natural language with terms from the problem domain.

In Regnell's thesis, he mentioned the new way of requirement engineering which is called UORE (Usage-Oriented Requirement Engineering). In the thesis, UCDA is extended with a synthesis phase, where use cases are formalized and integrated into a Synthesized Usage Model.

Regnell investigates the role of use case modeling in requirements engineering and its relation to system verification and validation [20]. His approach utilizes three levels for use case modeling. It allows a hierarchical structure and enables graphical representation at different abstraction levels. He investigates the possibility of integrating the two disciplines of use case modeling and statistical usage testing and discusses how they can be integrated to form a seamless transition from requirements models to test models for reliability certification. Two approaches are proposed for the integration of the use case model and the operational profile model. He also defines use case syntax and semantics.

Buhr 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 [3]. UCMs are a two dimensional map of cause-effect chains from points of stimuli through the system to points where responses are observed. UCMs consist of three primary constructs. Responsibilities are represented as dots, with the responsibility described by active natural language phrases. Causality is represented as a path that connects the dots, with start and end points that have associated pre- and post-conditions. Components are represented as simple boxes with associated responsibilities. UCMs can be refined through decomposition and partitioned by factoring.

UML 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 [25]: 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 notation to the UML. It represents the merger of SOMA [11], MOSES [12] and Firesmith [13]. A key principle behind OML is the notion of tasks and techniques. A task may be accomplished by one or more appropriate techniques. In OML, the relationship between a use case and a scenario is described in several ways as a specialization of a use case, an instance of a use case, and as a component of a use case. A use case links with objects via a participation association. OML also defines a specific category of scenario relationships, which are precedes, invokes and uses. In OML, capturing user requirements involves the use of task scripts, which are supported by a task-action grammar that consists of, a: Subject Verb

Direct Object Preposition Indirect Object (SVDPI). These task scripts can be organized into composition, classification and usage structures. Component scripts are derived through hierarchical task analysis. Side scripts deal with exceptions. Task scripts deal with business processes and not business functions. OML has explicit support for rule assertions that an object's operations must not violate. They take the form of pre-conditions and post-conditions and invariants.

3. Use Case Metamodel

3.1 Regnell's Use Case Metamodel

In Regnell's use case metamodel, he defines three levels for use case modeling. His model allows a hierarchical structure and enables graphical representation at different abstraction levels. The conceptual framework for the presented use case modeling approach [20] and their relations are illustrated in Figure 1

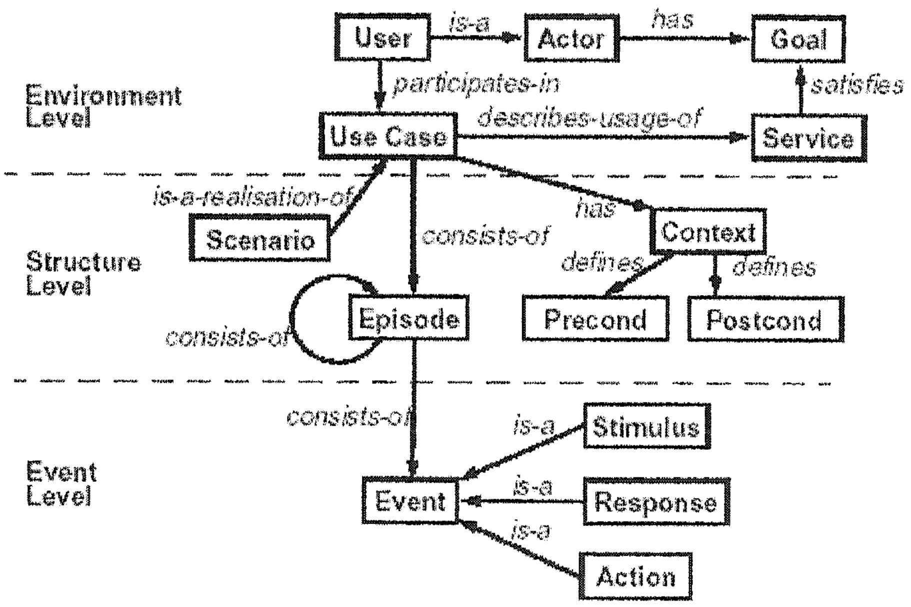


Figure 1. Regnell's use case metamodel

In figure 1, a use case model can be viewed on different abstraction levels.

1) The **environment level**

The **users** belong to the intended target system's environment and can be either humans or other software/hardware based systems. 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. Users can be of different types, called actors. A user is thus an instance of an actor. An **actor** (also called *user type* or *agent*) represents a set of users that have some common characteristics with respect to why and how they use the target system. Each actor has a set of goals, reflecting such common characteristics. **Goals** are objectives that users have when using the services of a target system. Thus, goals are used to categorize users into actors. The goals are described as patterns using general temporal operators such as *achieve*, *cease*, and *maintain* [7]. A **use case** represents a usage situation where one or more services of the target system are used by one or more actors with the aim to accomplish one or more goals.

2) The **structure level**

The structure level includes concepts that relates to the internal structure of use cases, such as different variants and parts of a use case. A use case may be divided into coherent parts, called **episodes.** The same episode can occur in many use cases. A **scenario** is a specific realization of a use case described as a sequence of a finite number of events. A scenario may either model a successful or an unsuccessful accomplishment of one or more goals. A use case may cover an unlimited number of scenarios as it may include alternatives and repetitions. A scenario, however, is a specific and bound realization of a use case, with all choices determined to one specific path. Every use case (and scenario) has a **context** that demarcates the scope of the use case and defines its **preconditions** (properties of the environment and the target system that need to be fulfilled in order to invoke the use case) and **post-conditions** (properties of the environment and the target system at use case termination). It is possible to have different degrees of scenario instantiation [18]; a completely instantiated scenario corresponds to a system usage trace, where the sequence of events is totally ordered and every parameter has a specific value. A scenario may also be on a slightly higher level, having symbolic names instead of specific parameter values.

3) The **event level**

The event level represents a lower abstraction level where the individual events are characterized. The lower abstraction level of uses cases, scenarios, and episodes includes **events** of three kinds: **stimuli** (messages from users to the target system), **responses** (messages from the target system to users), and **actions** (target system intrinsic events which are atomic in the sense that there is no communication between the target system and the users that participate in the use case). Stimuli and responses can have **parameters** that carry data to and from the target system. In order to express parameters, and also conditions on data, a use case model may be complemented by a data model.

3.2 Use Case Metamodel

For our own use case metamodel, we follow Mr. Rui's (Ph.D.) proposal [23] that is an extension of the Regnell's use case metamodel. Our metamodel also has three levels that are environment level, structure level and event level. Figure 2 shows our use case metamodel.

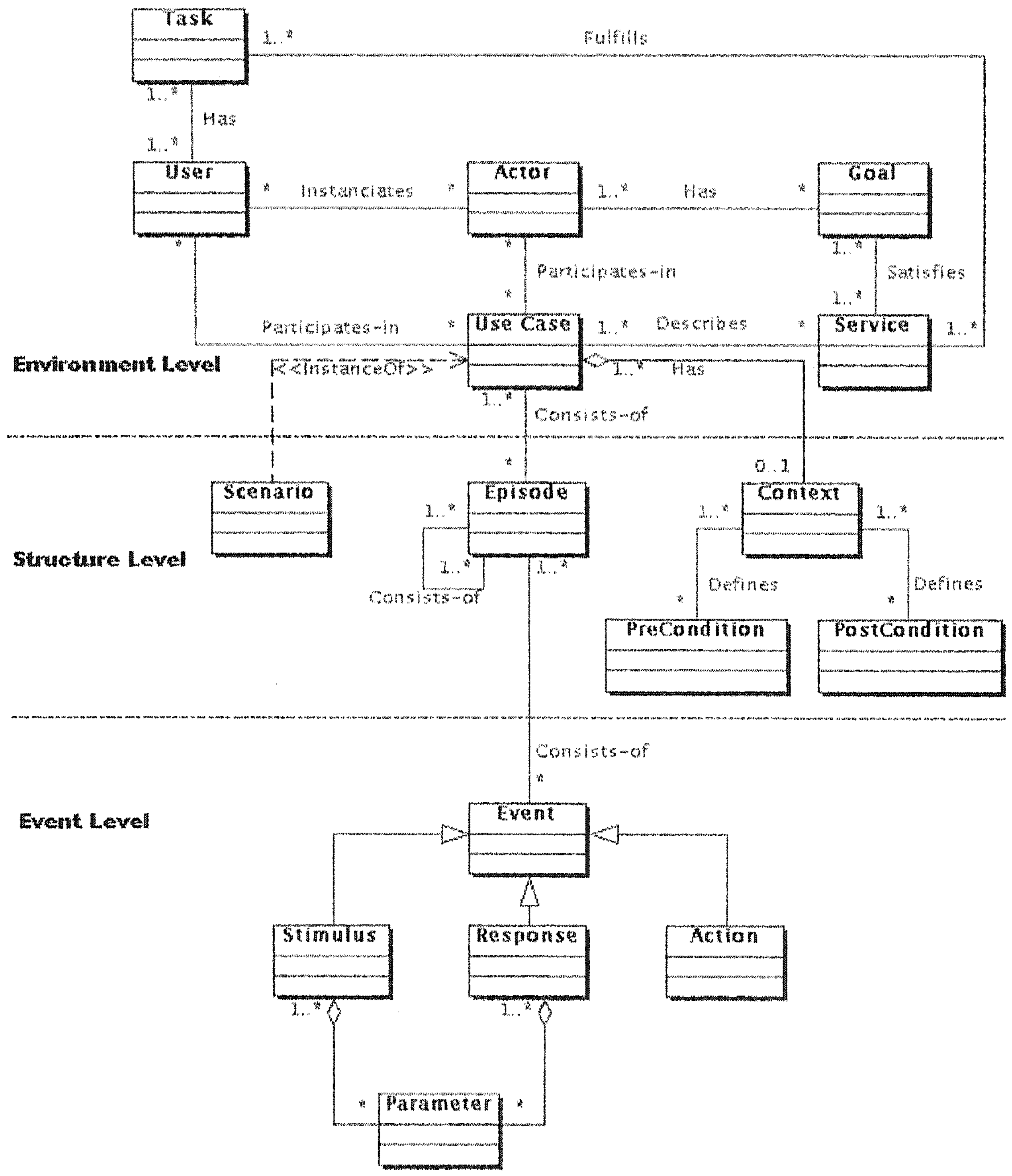


Figure 2. Use case metamodel

We define the most comprehensive use case relations in our use case metamodel, which cover UML, OML, SOMA, OOSE, and so on.

	Objectory Jacobson	**SOMA Graham**	**OML Firesmith**	**UML Rational**
Use Case Relationship	Uses Extends	Usage Composition Specialization	Invokes Precedes	Includes Extends Generalization

Table 1. Use case relationship

In the Table 1 we list the terminology on use case relations as described for Objectory by Jacobson [15], for SOMA by Graham [11], in the OPEN Modeling Language (OML) reference manual by Firesmith [8] and the Unified Modeling Language (UML) version 1.3 [14].

Among these relations, the uses-relation is semantically equivalent with an includes-relation. In fact, uses-relation is defined in UML version 1.1 [19], but it is replaced by includes-relation in UML version 1.3. Basically, the 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 intensions from the user's perspective. As Larry L.

Constantine puts: it makes it easier to validate the model with users and customers while also assuring that only one design will be developed [6]. Table 2 shown below is the summary of relationships defined in our use case metamodel.

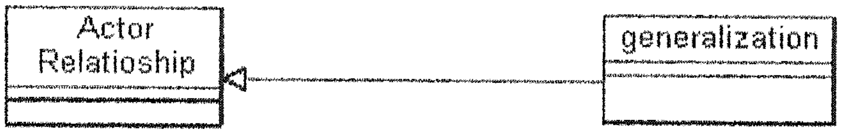
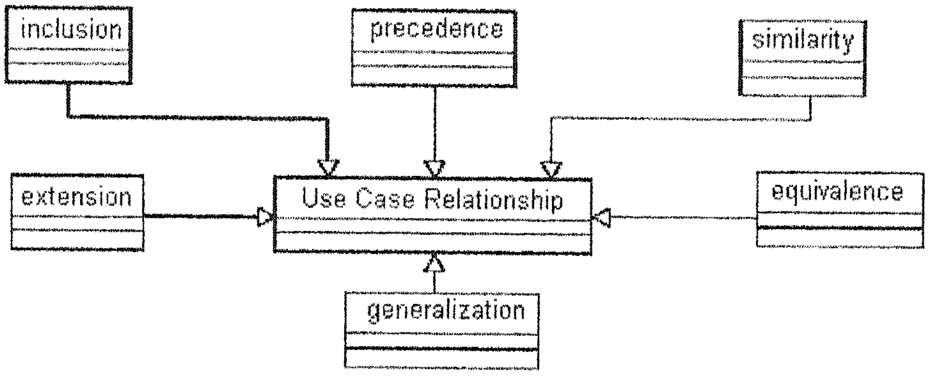
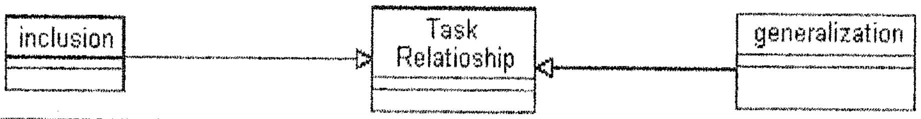
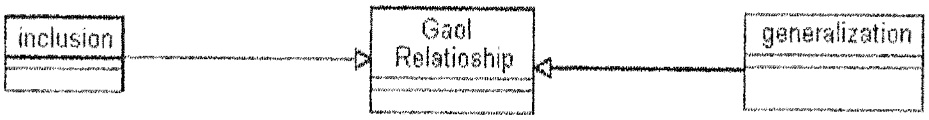
Actor	Generalization 
Use case	Inclusion Precedence Extension Similarity Generalization Equivalence  inclusion precedence similarity extension Use Case Relationship equivalence generalization
Task	Inclusion Generalization 
Goal	Inclusion Generalization 

Table 2. Summary of Relationships

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 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.

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 participates in all relationships of the parent use case. The child use case may also define new behavior sequences as well as add additional behavior 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.

An 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 intensions. 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 define 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 parent 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).

4. Refactoring

4.1 Introduction

The high cost of developing software motivates the reuse and evolution of existing software. Object-Oriented programming features such as abstraction, encapsulation, modularity and inheritance provide an infrastructure to software reuse. While code level reuse brings us with many benefits, people acknowledge that in the long run the design level reuse is more important.

Refactoring is the way of transform approach for iterative software development from changing a variable to changing the structure of the system with the behaviors of the system preserved. Refactoring has been used in source code for years. So applying refactoring to use case model is a very exciting approach. The principle of refactoring is to change the structure or relations between objects in the product without modifying the behaviors of the product. After refactoring users should feel no differences between the previous version and the version being refactorred. It's not easy to apply refactoring process on the requirement analysis because of the infinity of defining the requirements in a uniformed model. In code refactoring, tools can always find the exact code to be refactorred and the relations of the code as well. But in the requirement analysis stage, we have to use text, graphics and some other literal descriptions to describe what we get from the customers. When refactoring is applied to the requirement model, the tool should have enough intelligence and rules to make sure those refactorred requirements are depicted correctly without duplications and misses.

Following are brief introductions on some of the theories and people who contributed a lot to refactoring.

1. Opdyke

William Opdyke's Ph.D. thesis [17] is the first serious publication in refactorings. Inspired by the work of Banerjee and Kim for object-oriented database schema evolutions [1], his work is focused on automatic support for program restructuring (refactoring) to the object-oriented program. In his Ph.D. thesis, he identifies seven invariants required to preserve the behavior of C++. When a refactoring tends to violate an invariant, enabling conditions are added to ensure that the invariant is preserved. Seven invariants are Unique Superclass, Distinct Class Names, Distinct Member Names, Inherited Member Variables Not Redefined, and Compatible Signatures in Member Function Redefinition, Type-Safe Assignments, Semantically Equivalent References and Operations.

In his thesis he defines 26 low level refactorings as follows.

1. Creating a program Entity:

   (a) create_empty_class

   (b) create_member_variable

   (c) create_member_function

2. Deleting a program entity:

    (a) delete_unreferenced_class

    (b) delete_unreferenced_variable

    (c) delete_member_functions

3. Changing a program entity:

    (a) change_class_name

    (b) change_variable_name

    (c) change_member_function_name

    (d) change_type

    (e) change_access_control_mode

    (f) add_function_argument

    (g) delete_function_argument

    (h) reorder_function_arguments

    (i) add_function_body

    (j) delete_function_body

    (k) convert_instance_variable_to pointer

    (l) convert_variable_references_to_function_calls

    (m) replace_statement_list_with_function_call

    (n) inline_function_call

    (o) change_superclass

4. Moving a member variable:

    (a) move_member_variable_to_superclasses

(b) move_member_variable_to_subclasses

5. Intermediate level (composite) refactorings:

(a) abstract_access_to_member_variable

(b) convert_code_segment_to_function

(c) move_class

Each refactoring has some preconditions. Because these small refactorings are correct under certain preconditions, large changes that are composed solely of small refactorings must be correct. Therefore, refactoring can support software design and evolution by restructuring a program in the way that allows other changes to be made more easily. Complicated changes to a program can require both refactorings and additions. With these low level refactorings, a set of three high level refactorings which are more abstract are defined as follows.

1) Refactoring to generalize: creating an abstract superclass

2) Refactoring to specialize: subclassing, and simplifying conditions

3) Refactoring to capture: aggregation & components

2. Lance Tokuda

Based on Opdyke's research, Lance Tokuda goes further to evolve object-oriented designs with refactorings [27]. His research shows that all three kinds of design evolution, which are schema transformations, design pattern micro-architectures, and the hot-spot driven approach, are automatable with refactorings. Schema transformations perform many of the simple edits encountered when evolving class diagrams. They can be used alone or in combination to evolve object-oriented designs. The schema for an object-oriented database management system (OODBMS) looks like a class diagram of an object-oriented application. Among the 19 object-oriented database schema transformations, 12 transformations are implemented as automated refactorings by Tokuda. As with database schema transformations, refactorings have been shown to directly implement certain design patterns. Six patterns are automatable as refactorings ("Command, Composite, Decorator, Factory Method, Iterator and Singleton"). A number of patterns can be viewed as automatable program transformations applied to an evolving design. At least seven patterns from [10] can be viewed as a program transformation ("Abstract Factory, Adapter, Bridge, Builder, Strategy, Template Method and Visitor"). The hot-spot-driven-approach provides a comprehensive method for evolving designs to manage change in both data and functionality. Meta-patterns can be added to evolve designs. Most of them can be viewed as transformations from a simpler design. Refactorings automate the transition between designs granting designers the freedom to create simple frameworks and add patterns as needed when hot-spots are identified.

3. Donald Bradley Roberts

To make refactoring more practical, Donald Bradley Roberts dedicated to developing a commercial-grade tool, the Refactoring Browser for Smalltalk. In his Ph.D. thesis [21], he illustrates ways to make a refactoring tool both fast and reliable so that it is more useful. First of all, he extended the definition of refactoring presented by William F. Opdyke by adding post-conditions, which are assertions that a program must satisfy for the refactoring to be applied. Post-conditions describe how the assertions are transformed by the refactoring and they can be used for several purposes: to reduce the amount of analysis that later refactorings must perform, to derive preconditions of composite refactorings, and to calculate dependencies between refactorings. These techniques can be used in a refactoring tool to support undo, user-defined composite refactorings, and multi-user refactoring. He also defined a method to calculate the preconditions for composite refactorings. Each atomic refactoring within the composite has its own preconditions that must be true for the refactoring to be legal. Given a sequence of refactorings that make up a composite refactoring, it would be nice to be able to derive the preconditions for the composite refactoring from the individual refactorings that it comprises. In addition, he defined the dependency between refactorings based on commutativity. Large design changes can be composed of a sequence of smaller, more primitive refactorings. Since each step is a refactoring, the entire composition is also a refactoring. This composition property is very powerful in that it allows us to define and automate simpler, low-level refactorings, and then compose them into larger, complex refactorings. However, although the refactorings were initially specified in a sequence, they do not necessarily have to be performed in that sequence. Therefore, the real refactoring sequence should be defined in order to determine which refactoring occurs before others. Donald Roberts defined the formula for calculating the conditions under which any two refactorings may commute. Further, he implemented a set of applications that use the dependency information calculated from a chain of refactorings. Three of these applications are as follows. Undo, Parallelizing Refactorings and Merging refactorings in multi-user programming environments. Finally, he developed a scheme for using dynamically obtained information to perform refactorings, which can eliminate expensive static analysis by deferring the analysis to runtime.

4. Martin Fowler

Compared with Donald Roberts, Martin Fowler has been focusing on another direction, the refactoring process, to make refactoring more practical. With contributions of Kent Beck, John Brant, William Opdyke, and Donald Roberts, Martin Fowler published a book [9], which explains the principles and best practices of refactorings. Moreover, it provides a guideline for the refactoring process, where to start refactorings, when to start, when to stop, etc. The core of the book is a comprehensive catalog of refactorings. Each refactoring illustrates the motivation and mechanics of a proven code transformation.

5. Cascade Refactoring

Cascaded Refactoring [4] is a hybrid approach for the development and evolution of application 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 trace maps. The impact of the refactorings for one model is translated via trace maps to determine constraints on the refactorings of another model.

4.2 Refactorings on Use Case Model

Some refactoring rules are defined according to the definition of our use case metamodel. The refactoring rules are grouped into different categories. Each category consists of some detailed rules for creation, deletion or move, etc. As the first version of refactoring tool, the rules apply to the environment level and structure level only.

4.2.1 Refactoring Definitions

Refactoring rules defined for each level of our use case metamodel are shown in Table 3, Table 4 and Table 5 [24]. For those implemented in the current version, brief explanations are given.

Category	**Refactorings**	
Create	Create empty use case	Create empty user
	Create empty actor	Create empty task
	Create empty goal	Create empty service

Delete	Delete unreferenced use case	Delete unreferenced user
	Delete unreferenced actor	Delete unreferenced task
	Delete unreferenced goal	Delete unreferenced service
Change	Change use case name	Change task name
	Change actor name	Change goal name
	Change user name	Change service name
Move	Move goal	
Distribute	Decompose use case	Decompose task
	Decompose goal	
Compose	Include use case	Include goal
	Extend use case	Change parent use case
	Precede use case	Change parent actor
	Equal use case	Change parent task
	Include task	Change parent goal

Table 3. Refactorings on the Environment Level

In Table 3, move goal is to move a goal into a parent/child actor. The distribute category contains refactorings for distributing behavior. Decompose use case, for example, is to split one use case into two use cases. The compose category contains refactorings in establishing generalization, inclusion, extension, precedence and equivalence relationship. For example, change parent actor is to establish an inheritance relationship between actors.

Category	**Refactorings**	
Create	Create empty context	Create empty post-condition
	Create empty precondition	Create empty episode
Delete	Delete empty context	Delete empty post-condition
	Delete empty precondition	Delete unreferenced episode
Change	Change precondition name	Change episode name
	Change post-condition name	
Compose	Sequence episode	Except episode
	Alternative episode	Interrupt episode
	Repeat episode	Encapsulate episode
	Parallel episode	Change parent episode
Move	Move episode	Move episode to parent use case

Table 4. Refactorings on the Structure Level

In Table 4, the compose category contains refactorings in organizing episodes. For example, if there is time order between two episodes, sequence episode can be used to organize them. If one episode is an alternate of another, alternative episode can be used to implement this relationship. Move episode is to move an episode from one use case to another use case.

Category	**Refactorings**	
Stimulus	Change parent stimulus	Repeat stimulus
	Move stimulus	Except stimulus
	Sequence stimulus	Interrupt stimulus
Response	Change parent response	Repeat response
	Move response	Except response
	Sequence response	Interrupt response
Action	Change parent action	Move action

Table 5. Refactorings on the Event Level

In Table 5, there is a list of refactorings for organizing events. For example, change parent stimulus refactoring can be used to specialize/generalize one stimulus from others. The concept of cascaded refactoring is used to relate refactorings across different levels. The refactoring of one level determines constraints on the refactoring of another level via the traceability and alignment maps, which preserve their internal consistency and traceability. Our alignment maps are as follows:

1. the trace map from the environment level to the structure level;
2. the trace map from the structure level to the environment level;
3. the trace map from the structure level to the event level;
4. the trace map from the event level to the structure level;

4.2.2 Implemented Refactorings

In the first version of the tool, we implement the rules listed above. The arguments, precondition, post-condition and a brief description are given below for some of the implemented rules.

Refactorings on the Environment Level

Create Category

1. **create_empty_usecase**

   Description: create a new use case without any defined attributes (episodes, context, etc.)

   Arguments: new use case name

   Precondition: the name of the new use case is not used by an existing use case in the model.

2. **create_empty_actor**

   Description: create a new actor without any reference with other element (actor, use case, goal, etc.)

   Arguments: new actor name

   Precondition: the name of the new actor is not used by an existing actor in the model.

3. **create_empty_user**

   Description: create a new user without any reference with other element (actor, task, etc.)

Arguments: new user name

Precondition: the name of the new user is not used by an existing user in the model.

4. **create_empty_task**

   Description: create a new task without any reference with other element (user, task, etc.)

   Arguments: new task name

   Precondition: the name of the new task is not used by an existing task in the model.

5. **create_empty_goal**

   Description: create a new goal without any reference with other element (actor, goal, etc.)

   Arguments: new goal name

   Precondition: the name of the new goal is not used by an existing goal in the model.

6. **create_empty_service**

   Description: create a new service without any reference with other element (use case, etc.)

   Arguments: new service name

   Precondition: the name of the new service is not used by an existing service in the model.

7. **create_empty_context**

   Description: this refactoring defines a new context for U.

   Arguments: use case U

Precondition: there is no context defined in U.

Delete Category

1. **delete_unreferenced_usecase**

Description: this refactoring deletes an unreferenced use case.

Arguments: use case U

Precondition: at least one of the following two conditions is met.

1) U is empty and is not referenced by any other use case.

2) U is not referenced by any actor or any other use case.

Post-condition: after this refactoring, U is deleted from the model. This deletion will be cascaded to the structure level and event level. Suppose that U contains an episode E. If E is only referenced by U, E should be deleted as well. The same applies to event level.

2. **delete_unreferenced_actor**

Description: this refactoring deletes an unreferenced actor.

Arguments: actor A

Precondition: A is not referenced by any other actor, use case, user or goal

Post-condition: after this refactoring, A is deleted from the model.

3. **delete_unreferenced_user**

Description: this refactoring deletes an unreferenced user.

Arguments: user U

Precondition: U is not referenced by any use case, actor or task.

Post-condition: after this refactoring, U is deleted from the model.

4. **delete_unreferenced_task**

Description: this refactoring deletes an unreferenced task.

Arguments: task T

Precondition: T is not referenced by any other task or user.

Post-condition: after this refactoring, T is deleted from the model.

5. **delete_unreferenced_goal**

Description: this refactoring deletes an unreferenced goal.

Arguments: goal G

Precondition: G is not referenced by any other goal or actor.

Post-condition: after this refactoring, G is deleted from the model.

6. **delete_unreferenced_service**

Description: this refactoring deletes an unreferenced service.

Arguments: service S

Precondition: S is not referenced by any user or it is empty (it contains no use case).

Post-condition: after this refactoring, S is deleted from the model.

Change Category

1. **change_usecase_name**

Description: this refactoring changes a use case name.

Arguments: use case U, new name

Precondition: new name is not used by an existing use case in the model.

Post-condition: after this refactoring, U is changed to the new name.

2. **change_actor_name**

   Description: this refactoring changes an actor name.

   Arguments: Actor A, new name

   Precondition: new name is not used by an existing actor in the model.

   Post-condition: after this refactoring, A is changed to the new name.

3. **change_user_name**

   Description: this refactoring changes a user name.

   Arguments: user U, new name

   Precondition: new name is not used by an existing user in the model.

   Post-condition: after this refactoring, U is changed to the new name.

4. **change_task_name**

   Description: this refactoring changes a task name.

   Arguments: task T, new name

   Precondition: new name is not used by an existing task in the model.

   Post-condition: after this refactoring, T is changed to the new name.

5. **change_goal_name**

   Description: this refactoring changes a goal name.

   Arguments: goal G, new name

   Precondition: new name is not used by an existing goal in the model.

   Post-condition: after this refactoring, G is changed to the new name.

6. change_service_name

Description: this refactoring changes a service name.

Arguments: service S

Precondition: new name is not used by an existing service in the model.

Post-condition: after this refactoring, S is changed to the new name.

Compose Category

1. change_parent_usecase

Description: this refactoring changes the super use case of U1 to U2 (i.e. U2 becomes U1's parent use case).

Arguments: use case U1, use case U2

Precondition: following two conditions should be met

1) All episodes currently inherited in U1 will be identically inherited from U2.
2) There does not exist such an episode E that E exists in U2 but does not exist in U1

2. change_parent_actor

Description: this refactoring changes the super actor of A1 to A2 (i.e. A2 becomes A1's parent actor).

Arguments: actor A1, actor A2

Precondition: following two conditions should be met

1) All goals currently inherited in A1 will be identically inherited from A2
2) There does not exist such a goal G that G exists in A2 but does not exist in A1

3. **change_parent_goal**

Description: this refactoring changes the super goal of G1 to G2 (i.e. G2 becomes G1's parent goal).

Arguments: goal G1, goal G2

Precondition: following two conditions should be met

1) All actors currently inherited in G1 will be identically inherited from G2
2) There does not exist such an actor A that A exists in G2 but does not exist in G1

4. **change_parent_task**

Description: this refactoring changes the super task of T1 to T2 (i.e. T2 becomes T1's parent task)

Arguments: task T1, task T2

Precondition: following two conditions should be met

1) All users currently inherited in T1 will be identically inherited from T2
2) There does not exist such a user U that U exists in T2 but does not exist in T1

5. **include_usecase**

Description: this refactoring establishes an inclusion relationship between U1 and U2. U1 includes U2.

Arguments: usecase U1, U2

Precondition: every episode in U2 also exists in U1

6. **equal_usecase**

Description: this refactoring establishes an equivalence relationship between U1 and

U2.

Arguments: use case U1, U2

Precondition: U1 and U2 share the same set of episodes

Post-condition: after this refactoring, an equivalence relationship is established between U1 and U2.

Refactorings on the Structure Level

Create Category

1. **create_empty_episode**

Description: define a new episode without any defined event.

Arguments: new episode name

Precondition: the name of this episode does not clash with an already existing one in the model.

Delete Category

1. **delete_empty_context**

Description: this refactoring deletes an empty context for a use case.

Arguments: use case U

Precondition: the context in U is empty, that is, there isn't any defined precondition or postcondition

Post-condition: after this refactoring, the context is deleted from U.

2. **delete_unreferenced_episode**

Description: this refactoring deletes an unreferenced episode.

Arguments: episode E

Precondition: E is not referenced by any use case or any other episode

Post-condition: after this refactoring, E is deleted from the system. This deletion will be cascaded to the event level. (refer to delete_unreferenced_usecase refactoring)

Change Category

1. change_episode_name

Description: this refactoring changes an episode name.

Arguments: episode E, new name

Precondition: new name does not clash with an existing one in the model.

Post-condition: after this refactoring, E is changed to the new name.

Move Category

1. move_episode

Description: this refactoring moves episode E in use case U1 to use case U2

Arguments: use case U1 and its episode E1, use case U2

Precondition: U1 and U2 share same actors or actor of U2 is a super actor of the actor of U1. None of U1 and U2 is referenced by another use case. E1 is not referenced by any other episode in U1.

Post-condition: E1 is moved to U2. If U2 already contains E1 before this refactoring, keep only one E1 in U2. E1 is deleted from U1.

2. move_episodes_to_parent_usecase refactoring

Description: this refactoring moves a common episode E in use cases U1, U2, …, Un to use case U which is the parent use case of U1, U2, …, Un

Arguments: use case U and its sub use cases U1, U2, …, Un

Precondition: there is a common episode E which exists in U1, U2, …, Un

Post-condition: after this refactoring, there is one and only one E in U. There is no E in U1, U2, …, Un

5. Tool Design and Implementation

5.1 Drawlets Framework

Drawlets [22] is a two-dimensional graphics framework written in Java for building graphical applications that are inherently graphical. Its original framework, named HotDraw, was developed by K. Beck and W. Cunningham in Smalltalk. RoleModel Software Corporation ported it into Java and renamed it Drawlets.

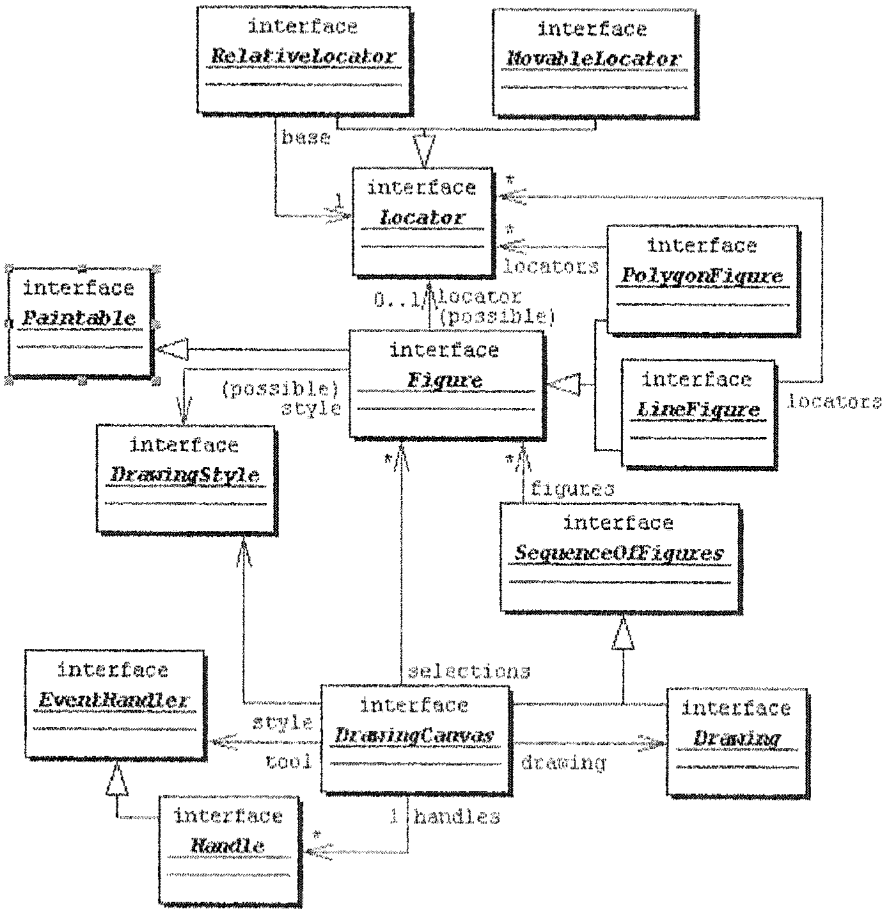


Figure 3. Major interfaces in Drawlets Framework

Drawlets is a 2D graphical drawing framework that provides flexible interfaces for vary applications. The major feature of the Drawlets framework is the drawing canvas that contains the tool and drawing components. The drawing component holds figures and listeners. Following is the class diagram for intended use of Drawlets fundamental roles

The tool component allows user to interact with figures to modify the attributes of a figure such as size and location. A special selection tool can select multiple figures and modify all of them.

As a framework, a) it is a collection of abstract classes and/or interfaces that provides an infrastructure common to a family of applications; b) it supports degrees of abstraction/interfaces and permits partial specifications; c) it is also a concrete realization in source code of a domain specific software architecture (graphical drawing). It can be applied in the reuse of code as well as requirements, architecture, design and documentation.

Drawlets supports several shapes, such as lines, freehand lines, triangles, rectangles, rounded rectangles, pentagons, polygons, ellipses, and text boxes. Each figure has a separate drawing tool. There are more than 100 classes, 35 interfaces and 40,000 lines of code and documentation. In the Drawlets framework there are many software design patterns, which provide great flexibility to the new application. The main interfaces used in Drawlets are shown below.

In our system, I use the Drawlets framework in the implementation of the environment level subsystem. Drawlets has been extended to define and show the elements such as use case, actor, etc. in graphics. The subsystem is very easy to extend for further modifications and new features since it is based on the Drawlets framework. For the structure level and event level, they don't use the graphical representations. Tree and list structures are used to show the episode and event instead of graphical elements. Because I work on some modules of the environment level, I will describe more about the design and implementation on that level than the other two levels.

5.2 System Design

5.2.1 Overview

The use case model refactoring tool is an MDI application. It uses Drawlets, tree and listbox in its GUI presentation. According to the objective, the tool can be divided into two parts:

1) definition subsystem

   In the definition subsystem, use case model can be defined in three levels that are mentioned in the use case metamodel and the definition data are stored in the XML format. The environment level is shown in graphics while the other two levels are shown in tree and list structures. The elements like use case, actor, etc. are extended from the basic shapes defined in the Drawlets framework.

2) refactoring subsystem

In the refactoring subsystem, refactoring rules can be applied to the metamodel defined by the definition subsystem in order to improve the model according to the changes on the system requirements. Refactoring rules can be used to modify the design of the model without changing its behaviors. The pre-conditions defined for the refactoring rules are used to validate the refactoring processing before it can be done. The refactoring method can be done only when all the pre-conditions are met.

5.2.2 Data Structure and Storage

XML is widely accepted for storage and information exchange [28]. It provides a number of positive attributes, such as tractability, extensibility, structure, openness, and independence between data and style. Document Type Definition (DTD) provides a grammar for creating XML document structure. We store our use case model using the XML format. Totally we use four XML files to store the use case layer, episode layer, event layer and goals separately.

a) Element definitions

Here I will introduce the definitions of elements used in the environment level in detail because my work focused on this level. For the other two levels, I will show some examples

1) Model

It is an aggregation of the entities, relations and some other elements.

Description:

```
<!DOCTYPE UsecaseModel [
   <!ELEMENT UsecaseModel (Usecases?, Actors?, Relationship?, Other?)?>
]>
```

2) Entity

There are two entities are defined in the model: use case and actor.

Description of use case and actor:

   id is unique for each use case and actor. Id is also used as a reference in the episode and event level. The position and size are also saved for being shown on the screen.

```
<!ELEMENT Usecases (Usecase)*>
<!ELEMENT Usecase(Name,Coordinate,Size*,Description,EpisodeID)>
   <!ATTLIST Usecase id CDATA #REQUIRED>
<!ELEMENT Actors (Actor)*>
<!ELEMENT Actor (Name,Coordinate,Size*,Description,UserID)>
   <!ATTLIST Actor id CDATA #REQUIRED>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Coordinate EMPTY>
   <!ATTLIST Coordinate x CDATA #REQUIRED y CDATA #REQUIRED>
<!ELEMENT Size EMPTY>
   <!ATTLIST Size width CDATA #REQUIRED height CDATA #REQUIRED>
<!ELEMENT Description (#PCDATA)>
```

<!ELEMENT EpisodeID (#PCDATA)>

<!ELEMENT UserID (#PCDATA)>

3) Relation

There are seven relations defined in this level: Association, Generalization, Inclusion, Extension, Similarity, Equivalence and Precedence.

Description of relations:

Relations must be adhering to use cases or actors.

```
<!ELEMENT Relationship(Association|Generalization|Inclusion|Extension|Similarity
|Equivalence|Precedence)*>
<!ELEMENT Association (Actor-Usecase*)?>
<!ELEMENT Generalization (Usecase-Usecase|Actor-Actor)*>
<!ELEMENT Inclusion (Usecase-Usecase*)?>
<!ELEMENT Extension (Usecase-Usecase*)?>
<!ELEMENT Similarity (Usecase-Usecase*)?>
<!ELEMENT Equivalence (Usecase-Usecase*)?>
<!ELEMENT Precedence (Usecase-Usecase*)?>
<!ELEMENT Usecase-Usecase (UsecaseRef1, UsecaseRef2)>
<!ELEMENT Actor-Actor (ActorRef1, ActorRef2)>
<!ELEMENT Actor-Usecase (ActorRef, UsecaseRef)>
<!ELEMENT UsecaseRef (#PCDATA)>
<!ELEMENT ActorRef (#PCDATA)>
```

```
<!ELEMENT UsecaseRef1 (#PCDATA)>
<!ELEMENT ActorRef1 (#PCDATA)>
<!ELEMENT UsecaseRef2 (#PCDATA)>
<!ELEMENT ActorRef2 (#PCDATA)>
```

4) Other

Elements defined below are complementary for the entities and relations. They are used to help complete the use case model. Three elements are defined: SystemBound, ServiceBound and TextLabel

Description of SystemBound, ServiceBound and TextLabel:

```
<!ELEMENT Other (SystemBound|ServiceBound|TextLabel)*>
<!ELEMENT SystemBound (Name, BoundRect, Description)>
<!ELEMENT ServiceBound (Name, BoundRect, Description, UsecaseEID)>
    <!ATTLIST ServiceBound id CDATA #REQUIRED>
<!ELEMENT TextLabel (BoundRect, Description)>
<!ELEMENT BoundRect EMPTY>
    <!ATTLIST BoundRect
          x CDATA #REQUIRED y CDATA #REQUIRED
          width CDATA #REQUIRED height CDATA #REQUIRED
    >
<!ELEMENT UsecaseEID (#PCDATA)>
```

b) Data storage

In the environment layer, there are five major entities: Model, Actor, Use case, Relation and Supplementary Entity.

1) A Model entity contains the model information and defines the possible sub-entity types in the model.
2) An Actor entity contains the actor information, such as name, communicating use case, parent actor, and so on. The id attribute of an Actor entity identifies the actor. Other attributes: x, y, width and height, are used by the use case editor to store the position and size of the actor figure.
3) A Usecase entity stores the use case information. Each Usecase entity has an identification attribute id. Other attributes are used by the use case editor, just like those of the Actor entity. An EpisodeID attribute stores the information of the referenced episode.
4) A Relation entity defines the relationship between two use cases, two actors or a use case and an actor. There are seven relations defined in this level: Association, Generalization, Inclusion, Extension, Similarity, Equivalence and Precedence. The ActorRef and UsecaseRef attributes refer to the id value of the corresponding actor and use case.
5) Supplementary Entities are used to help complete the model being compatible with UML semantics. There are three entities defined in the model: system bound, service bound and text label.

In the structure level, episode, subepisodes and subepisode are defined.

1) Episode is related to a use case. A use case can contain more episodes. There are two types of episode: primitive and composite. Primitive episode can be shared between use cases and be unique in the metamodel. Composite episode have four types: sequence, parallel, alternation and iteration

2) Subepisodes is used when composite episode is defined. It consists of subepisode.

3) Subepisode is included in Subepisodes which could be primitive or composite episode.

In the event level, primitiveevent is used to define one of stimuli, action and response.

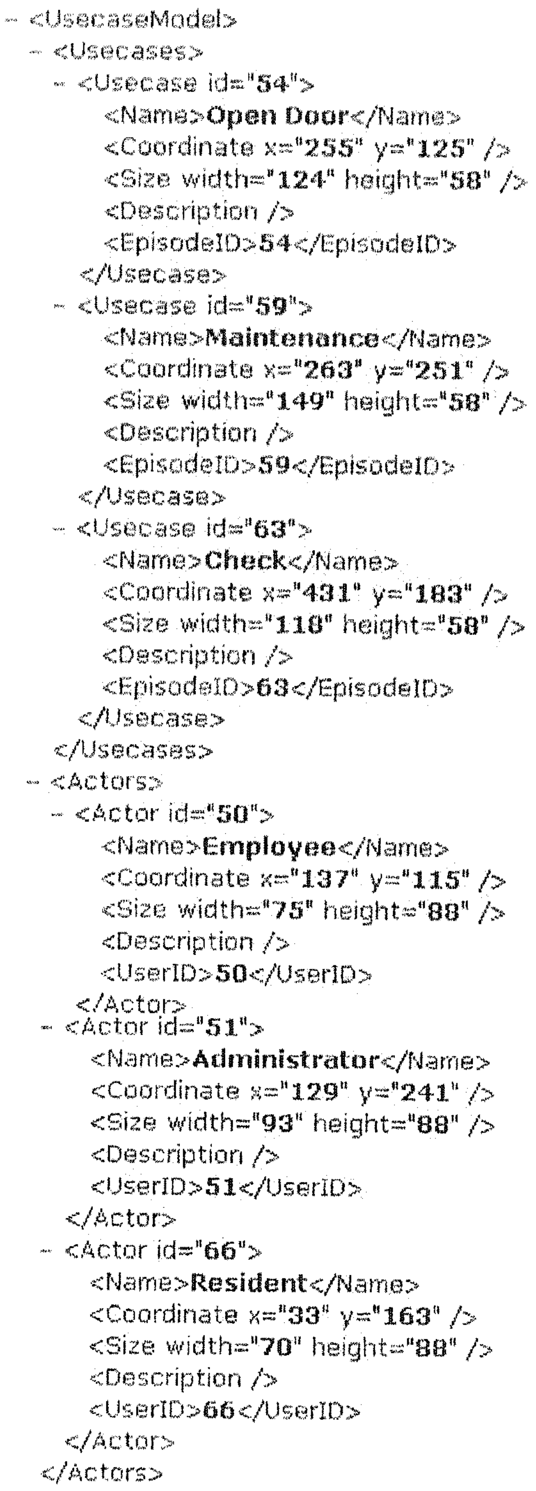
c) Data Storage Example

A simple example of use case metamodel is shown below. It contains all three layers that describe the system requirement of access control system (ACS).

Environment layer:

1) use case and actor definitions

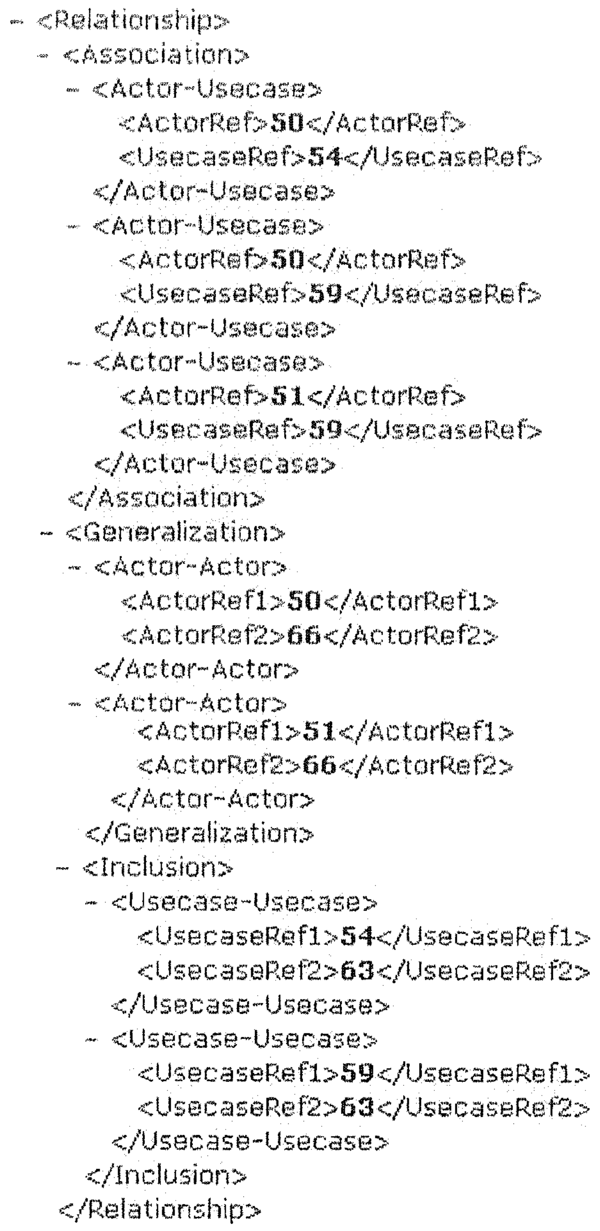
Entity's id is a unique id in the model. It is generated by the system automatically. Users cannot change the id. When the episode of the use case is defined, the id of a use case will be passed to the structure level as reference id.



```
- <UsecaseModel>
  - <Usecases>
    - <Usecase id="54">
        <Name>Open Door</Name>
        <Coordinate x="255" y="125" />
        <Size width="124" height="58" />
        <Description />
        <EpisodeID>54</EpisodeID>
      </Usecase>
    - <Usecase id="59">
        <Name>Maintenance</Name>
        <Coordinate x="263" y="251" />
        <Size width="149" height="58" />
        <Description />
        <EpisodeID>59</EpisodeID>
      </Usecase>
    - <Usecase id="63">
        <Name>Check</Name>
        <Coordinate x="431" y="183" />
        <Size width="118" height="58" />
        <Description />
        <EpisodeID>63</EpisodeID>
      </Usecase>
    </Usecases>
  - <Actors>
    - <Actor id="50">
        <Name>Employee</Name>
        <Coordinate x="137" y="115" />
        <Size width="75" height="88" />
        <Description />
        <UserID>50</UserID>
      </Actor>
    - <Actor id="51">
        <Name>Administrator</Name>
        <Coordinate x="129" y="241" />
        <Size width="93" height="88" />
        <Description />
        <UserID>51</UserID>
      </Actor>
    - <Actor id="66">
        <Name>Resident</Name>
        <Coordinate x="33" y="163" />
        <Size width="70" height="88" />
        <Description />
        <UserID>66</UserID>
      </Actor>
    </Actors>
```

Figure 4. XML representation of use case and actor

2) relation definitions

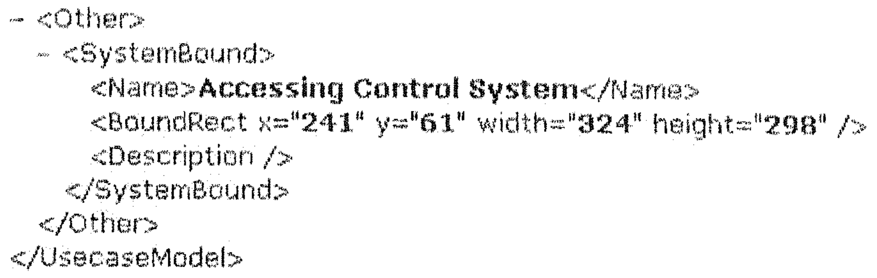


```
- <Relationship>
  - <Association>
    - <Actor-Usecase>
        <ActorRef>50</ActorRef>
        <UsecaseRef>54</UsecaseRef>
      </Actor-Usecase>
    - <Actor-Usecase>
        <ActorRef>50</ActorRef>
        <UsecaseRef>59</UsecaseRef>
      </Actor-Usecase>
    - <Actor-Usecase>
        <ActorRef>51</ActorRef>
        <UsecaseRef>59</UsecaseRef>
      </Actor-Usecase>
    </Association>
  - <Generalization>
    - <Actor-Actor>
        <ActorRef1>50</ActorRef1>
        <ActorRef2>66</ActorRef2>
      </Actor-Actor>
    - <Actor-Actor>
        <ActorRef1>51</ActorRef1>
        <ActorRef2>66</ActorRef2>
      </Actor-Actor>
    </Generalization>
  - <Inclusion>
    - <Usecase-Usecase>
        <UsecaseRef1>54</UsecaseRef1>
        <UsecaseRef2>63</UsecaseRef2>
      </Usecase-Usecase>
    - <Usecase-Usecase>
        <UsecaseRef1>59</UsecaseRef1>
        <UsecaseRef2>63</UsecaseRef2>
      </Usecase-Usecase>
    </Inclusion>
  </Relationship>
```

Figure 5. XML representation of relationship

System will check the entities linked by the relation. If the entities are not allowed to be linked by the relation that user wants to define, system will not generate the relation and show error to the user.

3) other definition

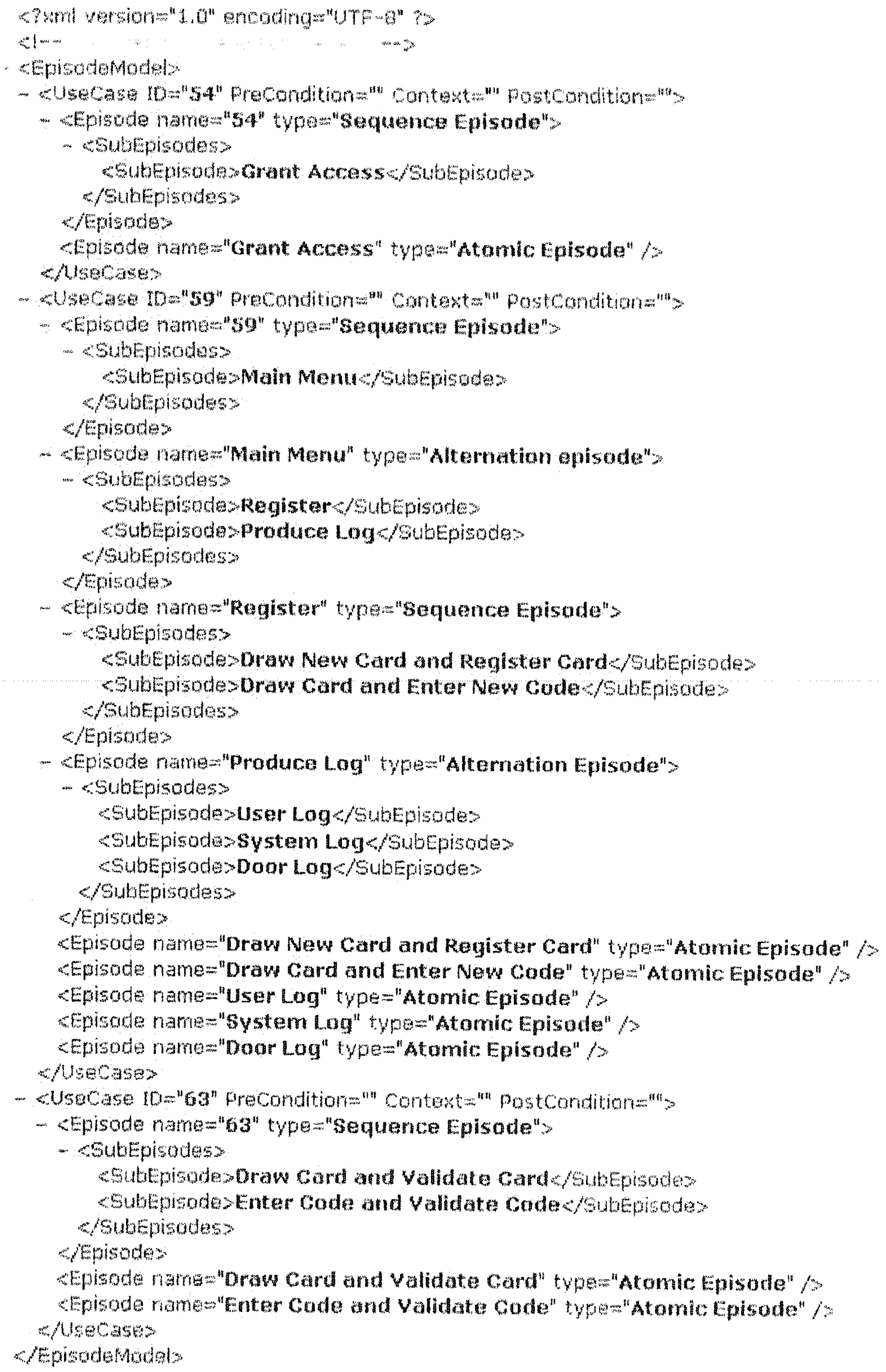


```
- <Other>
  - <SystemBound>
      <Name>Accessing Control System</Name>
      <BoundRect x="241" y="61" width="324" height="298" />
      <Description />
    </SystemBound>
  </Other>
</UsecaseModel>
```

Figure 6. XML representation of other elements

In the system, system bound, service bound and textbox are defined as Other entities. They are the supplementary of the model.

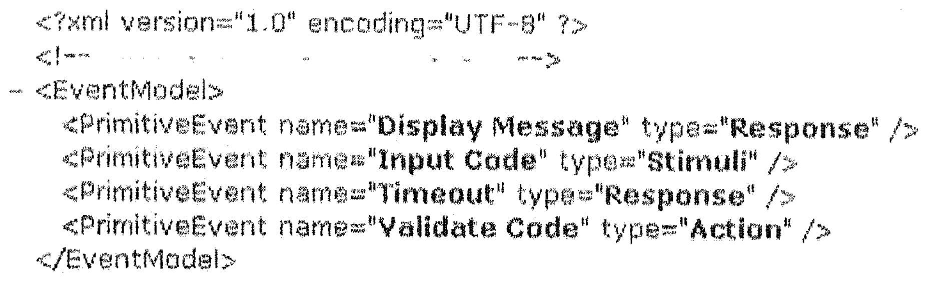
Structure layer:



```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- [illegible] -->
- <EpisodeModel>
 - <UseCase ID="54" PreCondition="" Context="" PostCondition="">
  - <Episode name="54" type="Sequence Episode">
   - <SubEpisodes>
       <SubEpisode>Grant Access</SubEpisode>
     </SubEpisodes>
   </Episode>
   <Episode name="Grant Access" type="Atomic Episode" />
  </UseCase>
 - <UseCase ID="59" PreCondition="" Context="" PostCondition="">
  - <Episode name="59" type="Sequence Episode">
   - <SubEpisodes>
       <SubEpisode>Main Menu</SubEpisode>
     </SubEpisodes>
   </Episode>
  - <Episode name="Main Menu" type="Alternation episode">
   - <SubEpisodes>
       <SubEpisode>Register</SubEpisode>
       <SubEpisode>Produce Log</SubEpisode>
     </SubEpisodes>
   </Episode>
  - <Episode name="Register" type="Sequence Episode">
   - <SubEpisodes>
       <SubEpisode>Draw New Card and Register Card</SubEpisode>
       <SubEpisode>Draw Card and Enter New Code</SubEpisode>
     </SubEpisodes>
   </Episode>
  - <Episode name="Produce Log" type="Alternation Episode">
   - <SubEpisodes>
       <SubEpisode>User Log</SubEpisode>
       <SubEpisode>System Log</SubEpisode>
       <SubEpisode>Door Log</SubEpisode>
     </SubEpisodes>
   </Episode>
   <Episode name="Draw New Card and Register Card" type="Atomic Episode" />
   <Episode name="Draw Card and Enter New Code" type="Atomic Episode" />
   <Episode name="User Log" type="Atomic Episode" />
   <Episode name="System Log" type="Atomic Episode" />
   <Episode name="Door Log" type="Atomic Episode" />
  </UseCase>
 - <UseCase ID="63" PreCondition="" Context="" PostCondition="">
  - <Episode name="63" type="Sequence Episode">
   - <SubEpisodes>
       <SubEpisode>Draw Card and Validate Card</SubEpisode>
       <SubEpisode>Enter Code and Validate Code</SubEpisode>
     </SubEpisodes>
   </Episode>
   <Episode name="Draw Card and Validate Card" type="Atomic Episode" />
   <Episode name="Enter Code and Validate Code" type="Atomic Episode" />
  </UseCase>
</EpisodeModel>
```

Figure 7. XML representation of episodes

Event layer:



```
  <?xml version="1.0" encoding="UTF-8" ?>
  <!-- [illegible] -->
- <EventModel>
    <PrimitiveEvent name="Display Message" type="Response" />
    <PrimitiveEvent name="Input Code" type="Stimuli" />
    <PrimitiveEvent name="Timeout" type="Response" />
    <PrimitiveEvent name="Validate Code" type="Action" />
  </EventModel>
```

Figure 8. XML representation of events

5.2.3 System Framework Design

Object-oriented methodology is used to design the system. According to the definition of the use case metamodel, the system has three levels: environment, structure and event level. Environment layer uses graphics to present the entities while tree and list structures are used in the other two levels to show the information of episode and event.

The Drawlets architecture is extended in three aspects.

a) The frame is extended in order to have a MDI interface.

b) The definition of object is extended in order to add new objects like use case, actor, service, etc.

c) Some new modules are added for use case, episode, event, goal and task.

I use Drawlets architecture in the implementation of the environment level. Drawlets is used as the basic framework of this level that provides the graphics supporting and event processing. Drawlets is also extended in order to implement the special needs from our metamodel. For example, interface Figure in Drawlets is extended to describe the entities like use case, actor, relations, etc. When the model is loaded into memory, Drawlets memory structure is used to store the runtime model structure. The XML file is used to store the model on the disk for future modification and data exchange.

Figure and DrawingCanvas are the two most important interfaces in Drawlets. All the Entities like actor, use case, etc. implement the Figure interface so that they can be added to the figure container without any further operations. DrawingCanvas is implemented as the figure container in order to keep all the entities at runtime.

1. System modules

The system structure of tool can be divided into four parts:

a) use case definition tool (environment level)

This tool contains drawing module, data reading/writing module, and refactoring module. By using this tool, we can define the use case as described in the UML specification. Use case, actor and service are used to describe the system functionalities, system users and services. Several relations are also defined to indicate the relationships between use case and actor.

b) goal description tool (environment level)

This tool includes goal definition, data display and refactoring supporting module. In our model, goal is used to describe what the actor can achieve after using system functions. It can be regarded as one of the ways to group users.

c) episode description tool (structure level)

This tool has episode definition, context/pre-condition/post-condition definition, data display, data reading/writing and refactoring supporting modules. Episode gives necessary validation when refactoring is to be done at use case layer. With this tool, we can give more detail description on the use case. The pre-condition and post-condition defined are the conditions must be met when the use case is invoked. The only way to go into episode layer is by selecting a use case in the use case layer. The data in episode layer are synchronized with use case layer. Moreover, episodes are shared by all the use cases in the same session that avoid duplication of episode.

d) event description tool (event level)

This tool is made up of event definition, data displaying, data reading/writing and refactoring modules. Similar to episode layer, it uses tree hierarchy structure to define and show the event. It also supports the synchronization between event layer and other layers. The events are shared by all the episodes as well.

Since I worked on the environment level, I will introduce more about this level other than the other two levels. In the environment level, it has four major parts:

a) Drawlets extension module

In this module, the main frame class in Drawlets is extended with some new features:

1) make the application to be a MDI application which can allow all the layers to be shown in the same frame

2) add loading and saving functions

3) define our own toolbar and palette

4) add automatically save all layers function

b) Model definition module

This module provides the capabilities of defining the use case view of the model including the use case, actor, relationship and service, etc. It also provides the way of stepping into structure level (from use case) and goal model (from actor).

c) Refactoring module

In this module, several refactoring rules are defined and applied to the use case model created by model definition module. It cooperates with episode and event layer to validate the pre-condition of refactoring rule. It also updates the use case metamodel according to the refactoring result.

d) JUnit testing module

Based on JUnit, some testing codes are added to verify the main functions defined in refactoring module. It gives an automatic and the fastest way of checking the refactoring functions especially after the modification on code. Several test suites are created which test different modules of the system.

2. System design diagrams

In the project, the use case editor is implemented based on the Drawlets framework. Several classes, such as SimpleDrawingCanvas, SimpleDrawing and BasicObservable, are modified. We add some new shapes such as actor, use case, and so on, based on the original shapes. Furthermore, we implement the attribute pane for some shapes, such as actor and use case, to define attributes of shapes. In order to store the use case model in the XML format, some classes are also added to the framework to parse and print the use case metamodel. The episode and event definition GUI are based on the Tree and List. They provide the way of defining primitive and composite episode and event. The Refactoring Tool GUI is embedded into the use case editor. Different refactoring rules can be chosen from the popup menu when different entities are selected. After a refactoring rules being selected, users will be guided to go through several steps to finish the refactoring process. In the steps, users may be asked to choose a use case or other entities, or to give a new value. The tool will validate the selection or input from user with the precondition of the refactoring rules. The refactoring rule can only be finished when all the preconditions are matched. Errors will be prompted when one of the preconditions isn't satisfied. Following are some use case and class diagrams of the tool.

a) Use case diagrams

Following are some use case diagrams for the refactoring tool.

Refactoring Tool Use Cases:

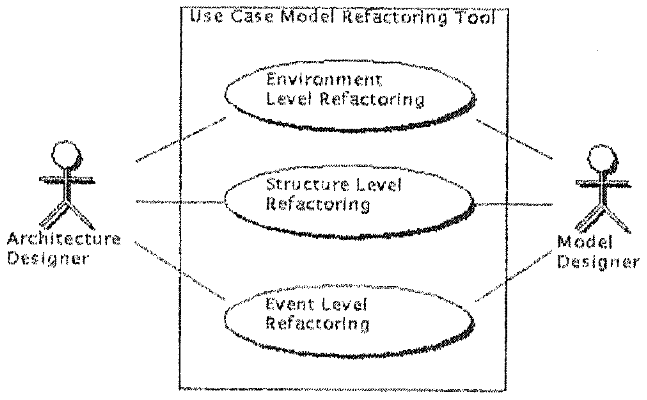

Use Case Model Refactoring Tool
Environment Level Refactoring
Structure Level Refactoring
Event Level Refactoring
Architecture Designer
Model Designer

Figure 9. Use case diagram of refactoring tool

Architecture designer and model designer will be the users of the system. They are allowed to create and modify use case model through several refactoring rules categories like creation, deletion, change and compose.

Environment Level Use Cases:

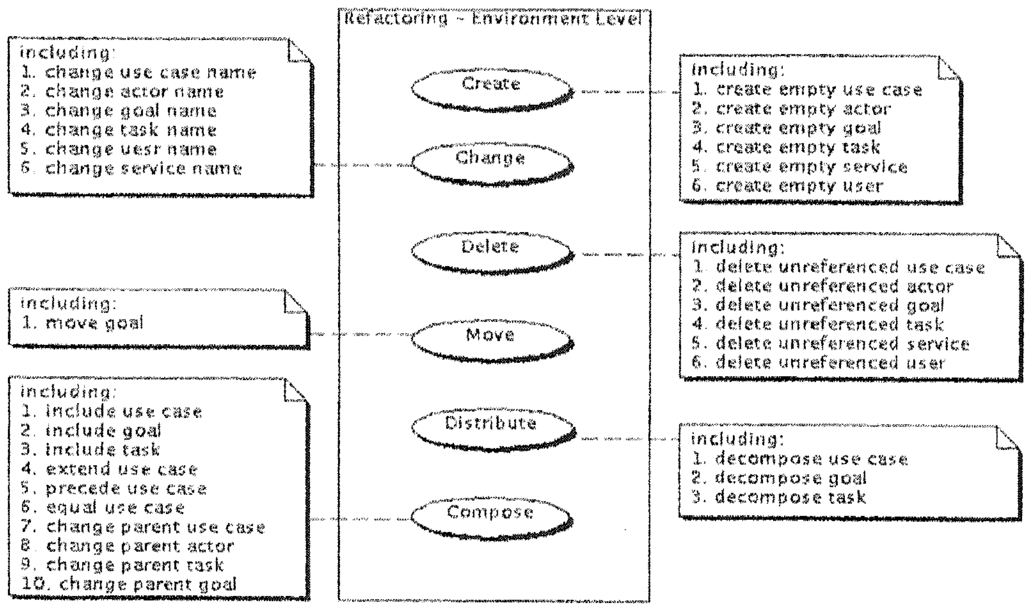


Figure 10. Use case diagram of environment level

Structure Level Use Cases:

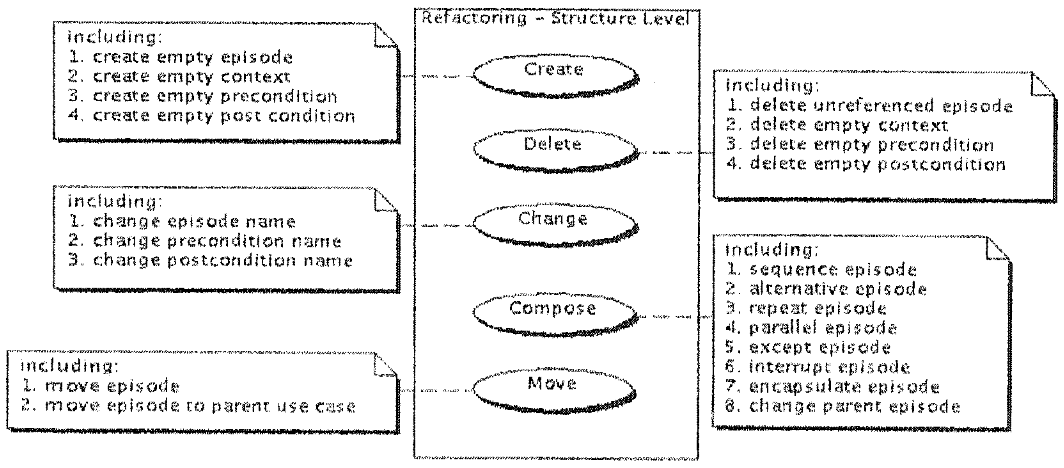


Figure 11. Use case diagram of structure level

Event Level Use Cases:

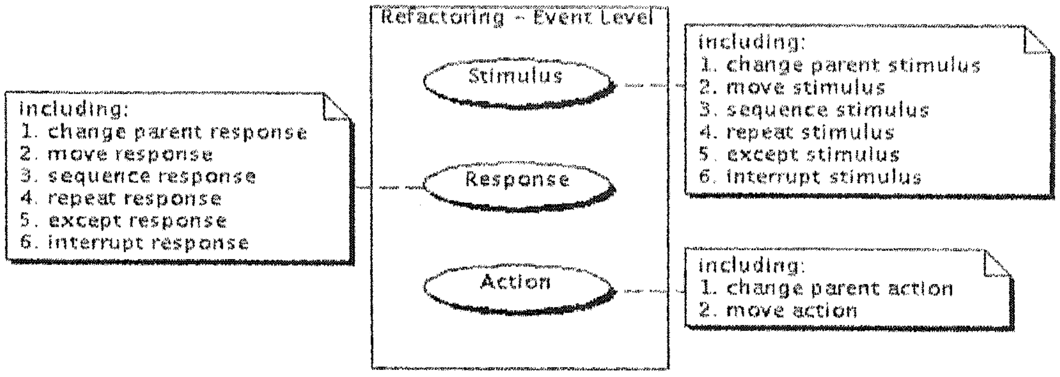


Figure 12. Use case diagram of event level

b) Class diagrams

I. System main packages

The whole system can be divided into three packages: Drawlets framework, metamodel and JUnit. The packages are shown in Figure 13.

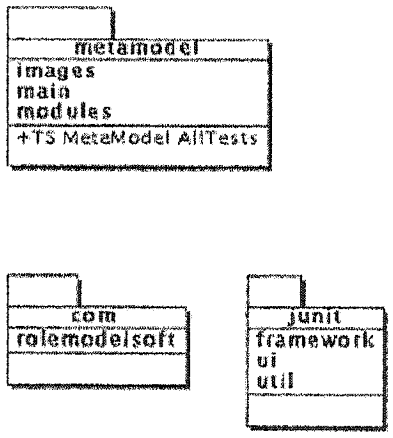


Figure 13. Packages of refactoring tool

Drawlets provides the base framework and give extra supports on the implementation of environment level subsystem. Metamodel is the aggregation of the implementations of all the three levels that are environment, structure and event. JUnit test helps us write testing code.

II. Use case model on environment level

Figure 14 shows the packages of the use case model on environment level.

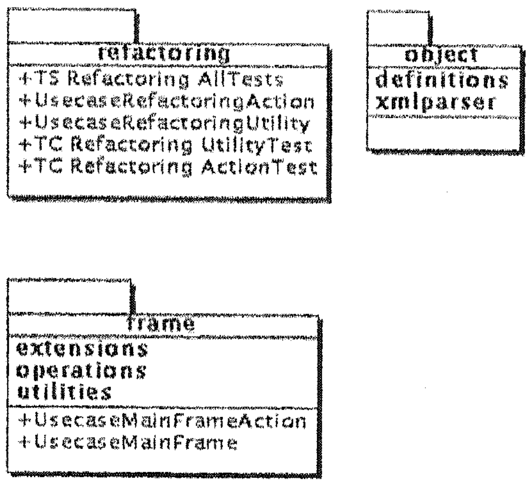


Figure 14. Packages of use case model on environment level

Frame package is inherited from the Drawlets framework; object package is used to define the entities and load/save data from/to XML files; refactoring package contains refactoring rules and unit test codes.

1) Object definition

Each object has two parts. One is the definition part that records the attributes of the object. The other part is the operation part that provides the way of how to create object, how to draw object and how to change object attributes. There are two types of objects: Entity (EntityBase and EntityBaseTool are the base classes) and Relation (RelationBase and RelationBaseTool are the base classes).

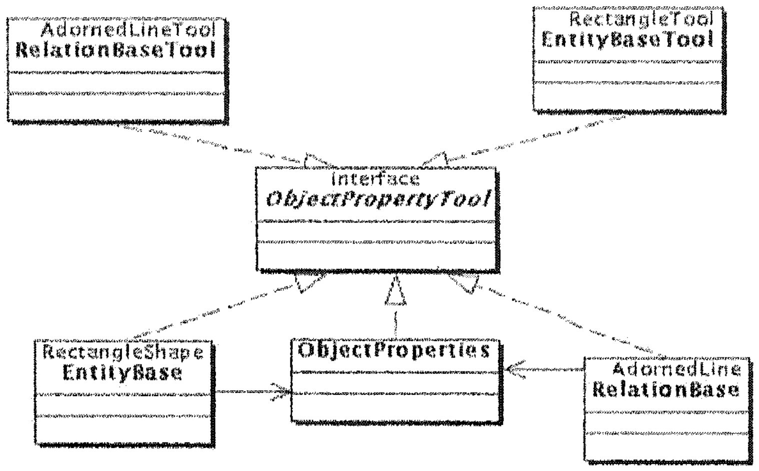


Figure 15. Class diagram of object definition

a) EntityBase

EntityBase is the base class for objects like Actor, ServiceBound, SystemBound and UseCase etc. EntityBase extends RectangleShape in Drawlets and defines the property collection of entity. In the subclass of EntityBase, how to draw an entity is defined in detail. The real values of properties are also given in the subclasses.

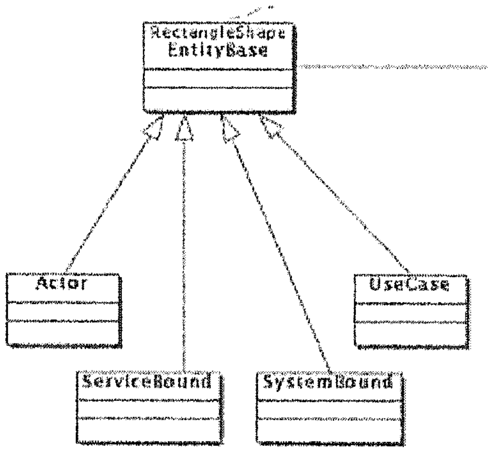


Figure 16. Class diagram of EntityBase

b) EntityBaseTool

EntityBaseTool is the base class for ActorTool, ServiceBoundTool, SystemBoundTool and UseCaseTool, etc. EntityBaseTool extends RectangleTool in Drawlets and defines the way of accessing properties of an entity. The subclass of EntityBaseTool defines how to create a new entity and react to various events from mouse or keyboard.

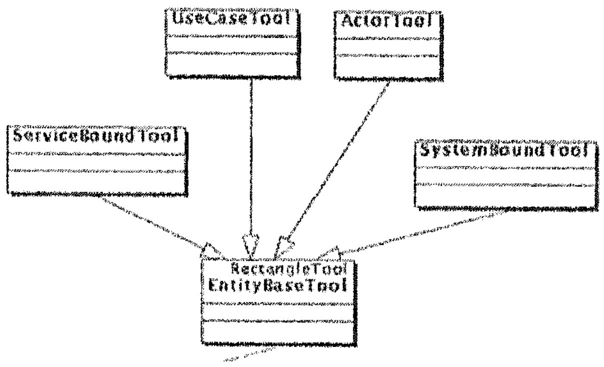


Figure 17. Class diagram of EntityBaseTool

c) RelationBase

RelationBase is the base class for relation objects like Generalization, Extension, Similarity, Equivalence, Association, Precedence and Inclusion, etc. RelationBase extends AdornedLine in Drawlets and define the properties collection of relation. RelationBase also defines the locators of a connection line between two entities. The subclass of RelationBase defines how to draw the connection line with specified text.

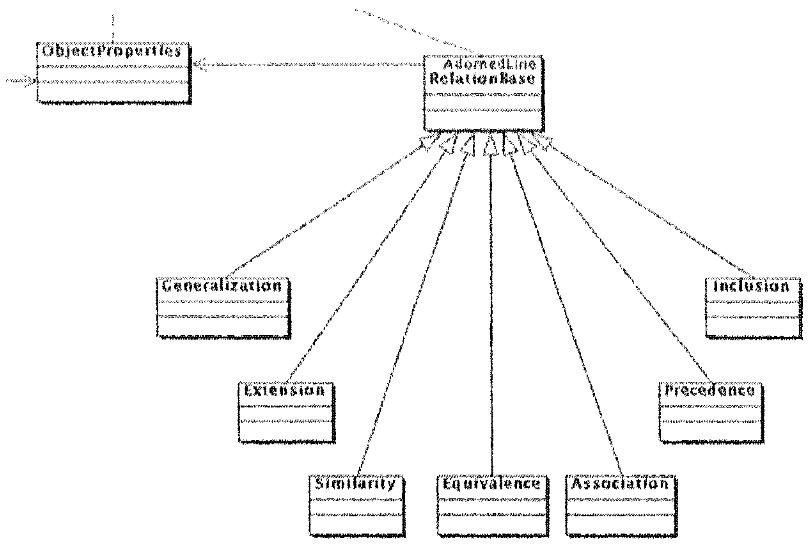


Figure 18. Class diagram of RelationBase

d) RelationBaseTool

RelationBaseTool is the base class for GeneralizationTool, ExtensionTool, SimilarityTool, EquivalenceTool, AssociationTool, PrecedenceTool and InclusionTool, etc. RelationBaseTool extends AdornedLineTool in Drawlets and define the way of accessing the properties of a relation. RelationBaseTool also defines how to check a relation can be created or not between two entities. For example, Generalization relation is not allowed between an Actor and a UseCase. The subclass of RelationBaseTool defines how to create a new relation when the relation is allowed between the two selected entities.

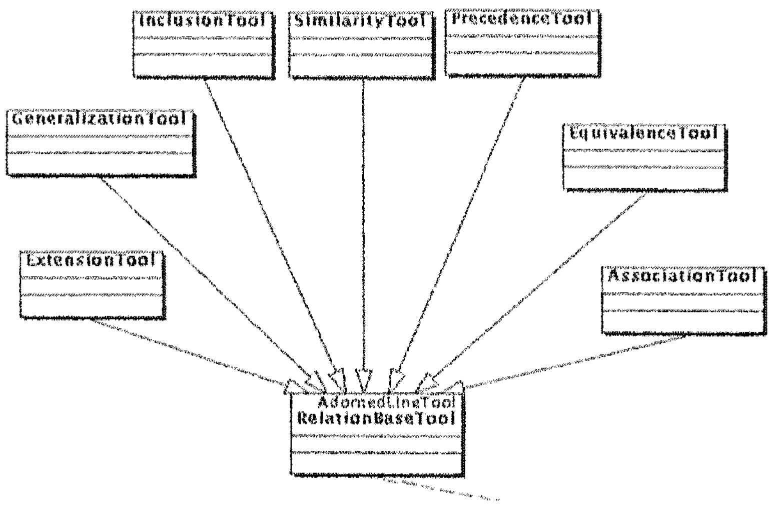


Figure 19. Class diagram of RelationBaseTool

2) Refactoring

Refactoring rules are implemented in two classes. One of the classes is UsecaseRefactoringAction that contains all the refactoring rules applied to the use cases at use case level. The other one is UsecaseRefactoringUtility which facilities the refactoring process and exposes interfaces to the other two levels for them getting entity information from use case level. JUnit TestCases are created in the Refactoring module so that each refactoring rules can be tested instantly and automatically within the JUnit framework application.

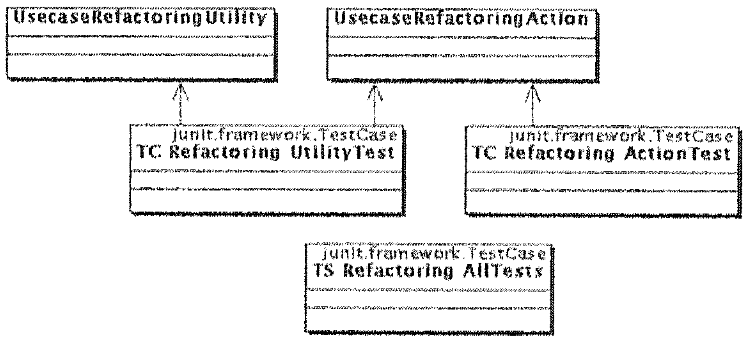


Figure 20. Class diagram of Refactoring

3) XML parser

XML parser is a package that provides the functionalities of loading and saving use case metamodel data from/to XML files. XMLProcessor is responsible for loading the data from XML files. It parses the XML files and translates the XML structures into use case metamodel elements structures and call the corresponding tools to create the entities or relations. XMLProcessor validates the data according to the DTD defined in the XML file. For saving the model, the classes, which implement the XMLGenerator interface, will save the DTD and the model data into XML file.

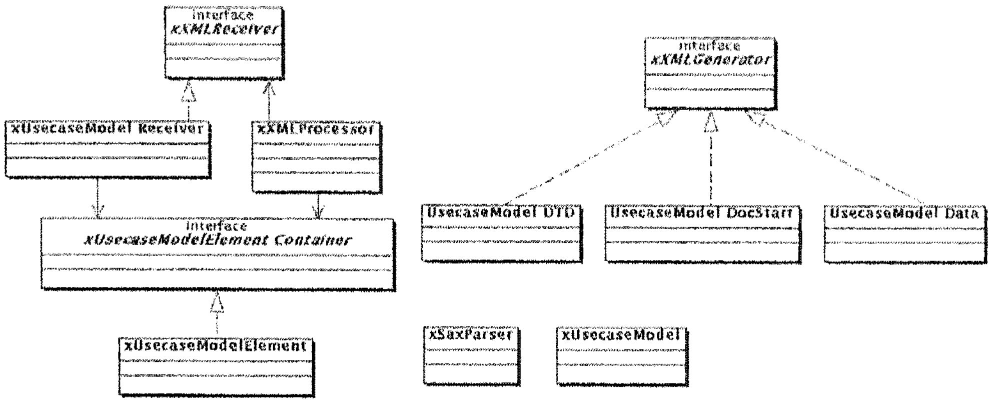


Figure 21. Class diagram of XML parser

III. Goal model

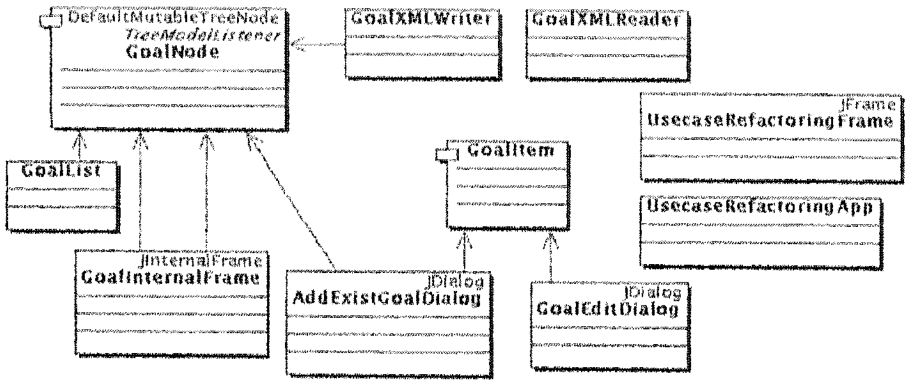


Figure 22. Class diagram of goal model

In the goal model, goals can be defined and saved through the GUI interface. Goals are related to a specified actor. While doing refactorings on actors, goals will be checked to validate the refactoring rule. Class diagram of goal model is shown in Figure 22.

IV. Episode model

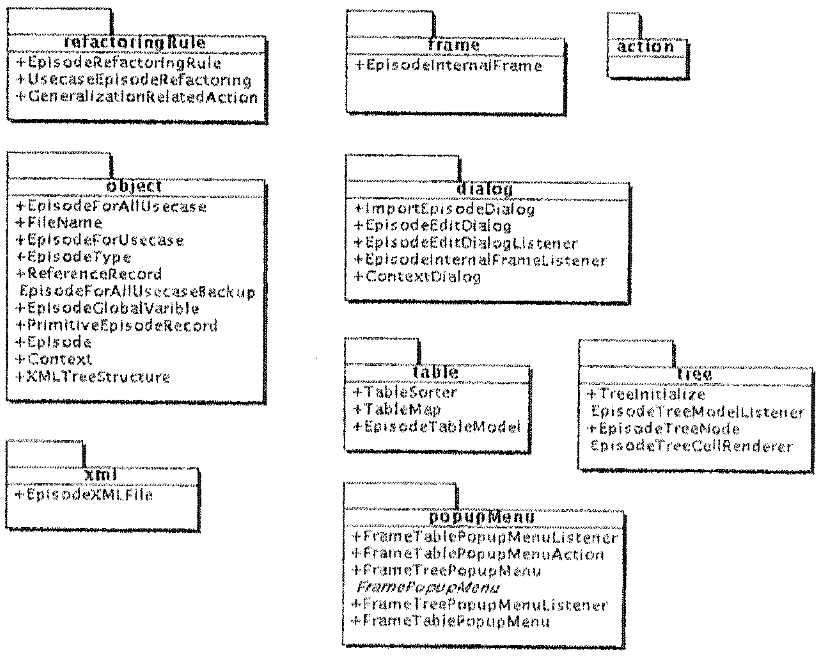


Figure 23. Class diagram of episode model

In the episode model, context, precondition and post-condition of use case can be defined. Users can also add primitive and composite episodes. Episodes can be shared between different episode models. Duplicated episodes are not allowed in the same episode model. Class diagram of episode model is shown in Figure 23.

V. Event model

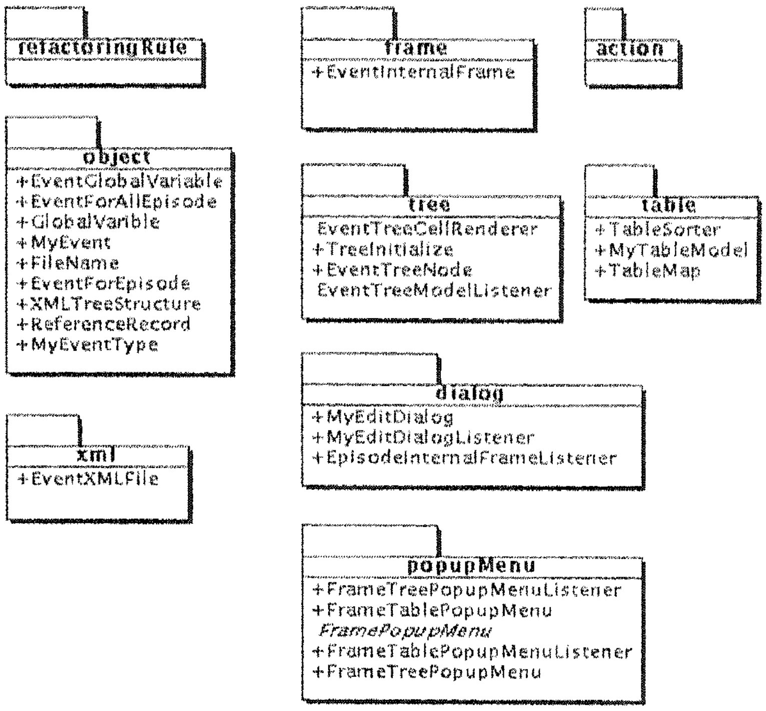
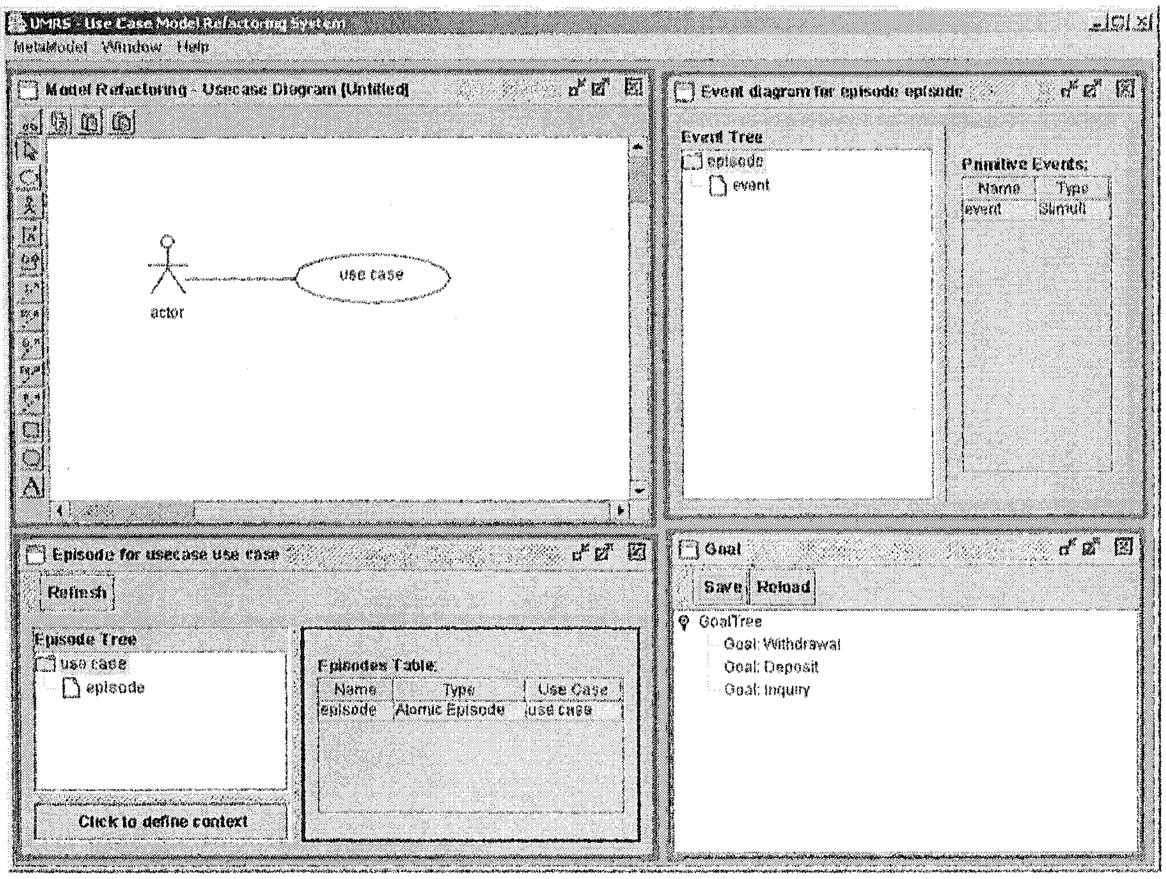


Figure 24. Class diagram of event model

In the event model, actions, responses and stimulus of a episode can be defined. Similar to episode model, all type of events can be shared between event models. Class diagram of event model is shown in Figure 24.

3. System screenshot

In the screenshot, different levels are shown in different sub-windows. Window on the top left represents the environment level. Window on the bottom left represents the structure level. Window on the top right represents the event level. Window on the bottom right represents the goal definition that belongs to the environment level.



Name	Type
event	Stimuli

Name	Type	Use Case
episode	Atomic Episode	use case

Figure 25. Screenshot of refactoring tool

This screenshot shows a very simple use case metamodel. There are only one use case and one actor defined in the environment level. In the structure level, a primitive episode of the use case is defined and an event of the episode is defined in the event level as well. Also three goals of actor are defined.

5.3 Summary

The refactoring tool is implemented according to the system design described above.

Users can use the tool to create use case model and apply the refactoring rules to the model. For understandability, more information of requirement of the targeting system can be described in the model with the tool that can help users have better understanding on the requirements. For reusability, the tool helps users reuse previous work easier because it provides both practical ways of refactoring on the model and the capability of extending the model to meet the changes on the requirements. For changeability, users can easily move information between different entities or levels with the refactoring rules in the model for decreasing the redundancy of the information. Those rules make further changes on the model become easier. For traceability, the tool also provides convenient ways of tracing the information of the model, for instance, users can get all the episodes of a use case at environment level, get the belonged use case of an episode at the structure level, or show all the events of a selected episode at structure level, etc.

Some issues are faced in implementation of my part of work of the tool. The Drawlets framework is based on AWT while all of us are working with Swing. I have to mix these two sets of GUI libraries in the implementation of the tool for making less modification on the Drawlets framework. MDI application is implemented to show not only different levels opened in different sub-windows but also different models opened in different sub-windows at the same time. XML parser is added to the tool for loading and saving model data because it is not provided by the Drawlets framework by default. This parser is also used in implementation of other levels.

6. Case Study

After the tool has been implemented, I use this tool to finish a case study that is a use case metamodel of ATM system. I chose the ATM (automatic teller machine) system because it's a very well known sample system that has been investigated thoroughly and referenced very often. The ATM system is used in lots of systems for functionalities testing and validation. With our tool, I designed and improved the use case metamodel of ATM system. The experience does validation on functionalities, feasibility and practicability of our metamodel and refactoring rules. A complete use case metamodel of ATM system is created in the end with full information. The metamodel is described in three levels and it can be used as the reference model for coding afterwards. The model is also easily altered by using the refactoring rules when the requirements change. I will give a brief description of the system requirements of ATM first. Then I will show the whole process of creating the use case metamodel according to the requirements. The process starts from a very simple definition of the requirements. Then by applying the refactoring rules to the metamodel according to changes on requirements, the metamodel is changed and becomes more complex.

6.1 Requirements of ATM System

The following description of the requirements of the ATM system is based on Prof.

Russell C. Bjork's ATM Simulation example used in the course called "Object-Oriented Software Development" [2].

The ATM system provides the ability for banking customers to deposit, withdraw, and transfer funds. It also allows them to inquire about account balances. The commonly used ATM unit is built into a wall and provides a small screen with buttons on both sides and a keypad below. Some problems with this ATM is that the lines and arrows usually do not line up with the correct buttons and because the key pad is so far away from the screen, the user has trouble noticing some of the buttons like clear and cancel.

Design the software to support a computerized backing network including both human cashiers and automatic teller machines (ATMs) to be shared by a consortium of banks. Each bank provides its own computers to maintain its own accounts and process transactions against them. Cashier stations are owned by individual banks and communicate directly with their own bank's computers. Human cashiers enter account and transaction data. Automatic teller machines communicate with a central computer which clears transactions with the appropriate banks. An automatic teller machine accepts a cash card, interacts with the user, communicates with the central system to carry out the transaction, dispenses cash, and prints receipts. The system requires recordkeeping and security provisions, the system must handle concurrent accesses. The banks will provide their own software for their own computers; you are to design the software for the ATMs and the network. The cost of the shared system will be apportioned to the banks according to the number of customers with cash cards.

The ATM is used by customers of a bank. Each customer has two accounts: a checking account and a savings account. Each customer has a customer number and a Personal Identification Number (PIN). Both must be typed into the simulation to gain access to the accounts. Once they have gained access, the customer can select an account (checking or savings). The balance of the selected account is displayed (initially zero). Then the customer can deposit and withdraw money. The application terminates when the user selects exit rather than an account. Since this is a simulation, the ATM does not actually communicate with the bank. It simply loads a list of customer numbers and PINs from a data file.

The software to be designed will control a simulated automated teller machine (ATM) having a magnetic stripe reader for reading an ATM card, a customer console (keyboard and display) for interaction with the customer, a slot for depositing envelopes, a dispenser for cash (in multiples of $20), a printer for printing customer receipts, and a key-operated switch to allow an operator to start or stop the machine. The ATM will communicate with the bank's computer over an appropriate communication link. (The software on the latter is not part of the requirements for this problem.)

The ATM will service one customer at a time. A customer will be required to insert an ATM card and enter a personal identification number (PIN) - both of which will be sent to the bank for validation as part of each transaction. The customer will then be able to perform one or more transactions. The card will be retained in the machine until the customer indicates that he/she desires no further transactions, at which point it will be returned - except as noted below.

The ATM must be able to provide the following services to the customer:

1. A customer must be able to make a cash withdrawal from any suitable account linked to the card, in multiples of $20.00. Approval must be obtained from the bank before cash is dispensed.
2. A customer must be able to make a deposit to any account linked to the card, consisting of cash and/or checks in an envelope. The customer will enter the amount of the deposit into the ATM, subject to manual verification when the envelope is removed from the machine by an operator. Approval must be obtained from the bank before physically accepting the envelope.
3. A customer must be able to make a transfer of money between any two accounts linked to the card.
4. A customer must be able to make a balance inquiry of any account linked to the card.
5. A customer must be able to abort a transaction in progress by pressing the Cancel key instead of responding to a request from the machine.

The ATM will communicate each transaction to the bank and obtain verification that it was allowed by the bank. Ordinarily, a transaction will be considered complete by the bank once it has been approved. In the case of a deposit, a second message will be sent to the bank indicating that the customer has deposited the envelope. (If the customer fails to deposit the envelope within the timeout period, or presses cancel instead, no second message will be sent to the bank and the deposit will not be credited to the customer.)

If the bank determines that the customer's PIN is invalid, the customer will be required to re-enter the PIN before a transaction can proceed. If the customer is unable to successfully enter the PIN after three tries, the card will be permanently retained by the machine, and the customer will have to contact the bank to get it back.

If a transaction fails for any reason other than an invalid PIN, the ATM will display an explanation of the problem, and will then ask the customer whether he/she wants to do another transaction.

The ATM will provide the customer with a printed receipt for each successful transaction, showing the date, time, machine location, type of transaction, account(s), amount, and ending and available balance(s) of the affected account ("to" account for transfers).

The ATM will have a key-operated switch that will allow an operator to start and stop the servicing of customers. After turning the switch to the "on" position, the operator will be required to verify and enter the total cash on hand. The machine can only be turned off when it is not servicing a customer. When the switch is moved to the "off" position, the machine will shut down, so that the operator may remove deposit envelopes and reload the machine with cash, blank receipts, etc.

The ATM will also maintain an internal log of transactions to facilitate resolving ambiguities arising from a hardware failure in the middle of a transaction. Entries will be made in the log when the ATM is started up and shut down, for each message sent to the

Bank (along with the response back, if one is expected), for the dispensing of cash, and for the receiving of an envelope. Log entries may contain card numbers and dollar amounts, but for security will never contain a PIN.

6.2 Creation of ATM Use Case Model

According to the requirements described above, I use the tool to define the use case model of the ATM system in the following steps.

1. Create a new metamodel. Add three use cases: 'Withdraw', 'Deposit' and 'Inquire' and one actor 'Customer' to the model. 'Customer' uses those three use cases.

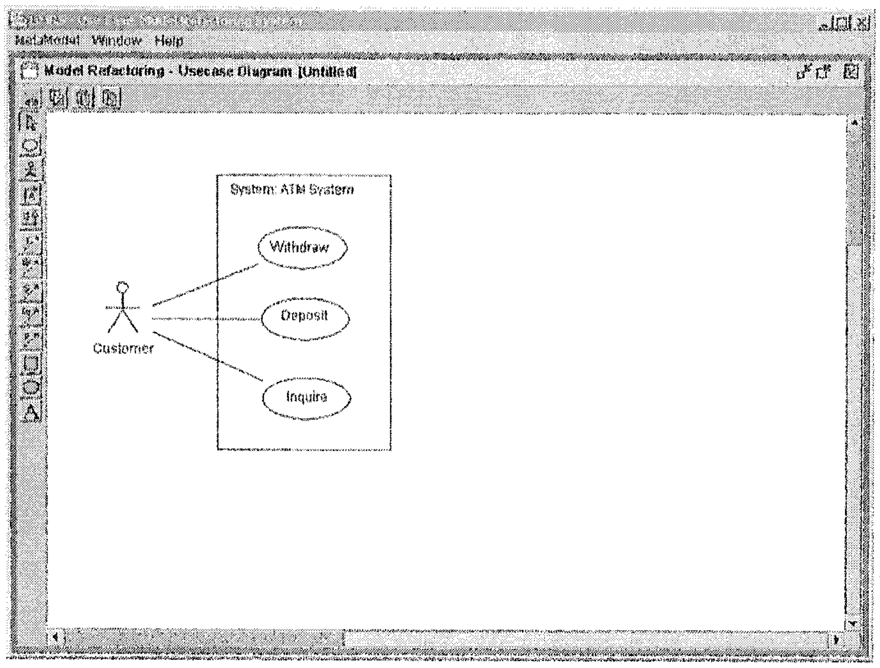
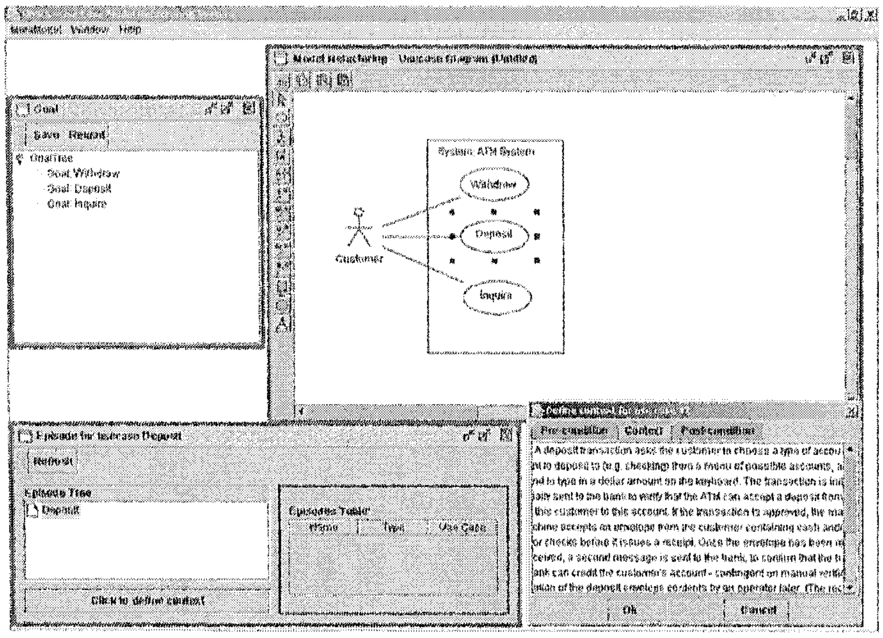


Figure 26. Use case model of ATM system after Step 1

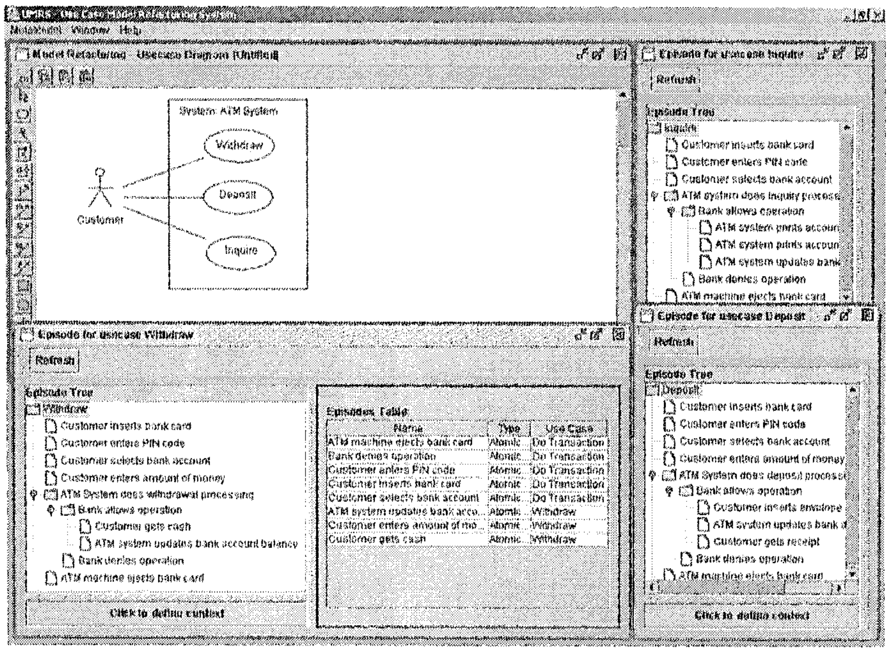
2. Define goals of the actor 'Customer' and the pre-condition, context and post-condition of the use cases 'Withdraw', 'Deposit' and 'Inquire'



A deposit transaction asks the customer to choose a type of accou
nt to deposit to (e.g. checking) from a menu of possible accounts, a
nd to type in a dollar amount on the keyboard. The transaction is init
ially sent to the bank to verify that the ATM can accept a deposit from
this customer to this account. If the transaction is approved, the ma
chine accepts an envelope from the customer containing cash and/
or checks before it issues a receipt. Once the envelope has been re
ceived, a second message is sent to the bank, to confirm that the b
ank can credit the customer's account - contingent on manual verific
ation of the deposit envelope contents by an operator later. (The rec

Figure 27. Use case model of ATM system after Step 2

3. Define episodes of use cases 'Withdraw', 'Deposit' and 'Inquire'


UMRS - Use Case Model Refactoring System
MetaModel Window Help
Model Refactoring - Usecase Diagram (Untitled)
System: ATM System
Withdraw
Deposit
Inquire
Customer
Episode for usecase Withdraw
Episode for usecase Inquire
Episode for usecase Deposit
Refresh
Episode Tree
Click to define context

- Withdraw
  - Customer inserts bank card
  - Customer enters PIN code
  - Customer selects bank account
  - Customer enters amount of money
  - ATM System does withdrawal processing
    - Bank allows operation
      - Customer gets cash
      - ATM system updates bank account balance
    - Bank denies operation
  - ATM machine ejects bank card

Episodes Table

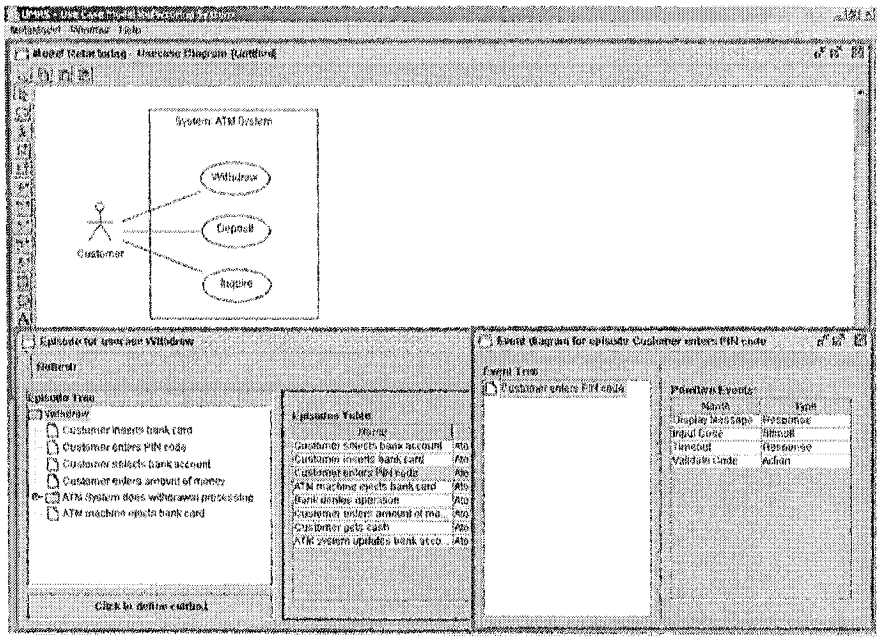
Name	Type	Use Case
ATM machine ejects bank card	Atomic...	Do Transaction
Bank denies operation	Atomic...	Do Transaction
Customer enters PIN code	Atomic...	Do Transaction
Customer inserts bank card	Atomic...	Do Transaction
Customer selects bank account	Atomic...	Do Transaction
ATM system updates bank acco...	Atomic...	Withdraw
Customer enters amount of mo...	Atomic...	Withdraw
Customer gets cash	Atomic...	Withdraw

- Inquire
  - Customer inserts bank card
  - Customer enters PIN code
  - Customer selects bank account
  - ATM system does inquiry processi
    - Bank allows operation
      - ATM system prints accoun
      - ATM system prints accoun
      - ATM system updates bank
    - Bank denies operation
  - ATM machine ejects bank card

- Deposit
  - Customer inserts bank card
  - Customer enters PIN code
  - Customer selects bank account
  - Customer enters amount of money
  - ATM System does deposit processi
    - Bank allows operation
      - Customer inserts envelope
      - ATM system updates bank a
      - Customer gets receipt
    - Bank denies operation
  - ATM machine ejects bank card

Figure 28. Use case model of ATM system after Step 3

4. Define events of episodes of use cases 'Withdraw', 'Deposit' and 'Inquire'


System: ATM System
Withdraw
Deposit
Inquire
Customer
Episode for usecase Withdraw
Refresh
Episode Tree
Withdraw
Customer inserts bank card
Customer enters PIN code
Customer selects bank account
Customer enters amount of money
ATM System does withdrawal processing
ATM machine ejects bank card
Click to define context
Event diagram for episode Customer enters PIN code
Event Tree
Customer enters PIN code

Episodes Table

Name	
Customer selects bank account	Ato
Customer inserts bank card	Ato
Customer enters PIN code	Ato
ATM machine ejects bank card	Ato
Bank denies operation	Ato
Customer enters amount of mo...	Ato
Customer gets cash	Ato
ATM system updates bank acco...	Ato

Pointers Events:

Name	Type
Display Message	Response
Input Code	Stimuli
Timeout	Response
Validate Code	Action

Figure 29. Use case model of ATM system after Step 4

5. Transaction is needed in banking system. The current three use cases should support transactions. Also shared episodes are found in the current three use cases. So a super use case 'Do Transaction' is created to be the parent use case of those three use cases. Those shared episodes will be moved to super use case 'Do Transaction'. A new service is also created to include all the use cases.

    1) Use 'Create Empty Service' to create service 'Operation'

    2) Add use cases 'Withdraw', 'Deposit' and 'Inquire' into service 'Operation'.

    3) Use 'Create empty use case' to create a use case 'Do Transaction'. Use case 'Do Transaction' is also defined in the service 'Operation'

    4) Use 'Change Parent Use Case' to set use case 'Do Transaction' as the parent of use cases 'Withdraw', 'Deposit' and 'Inquire'.

    5) Use 'Move Episode to Parent Use Case' to move common episodes in use cases

'Withdraw', 'Deposit' and 'Inquire' to use case 'Do Transaction'

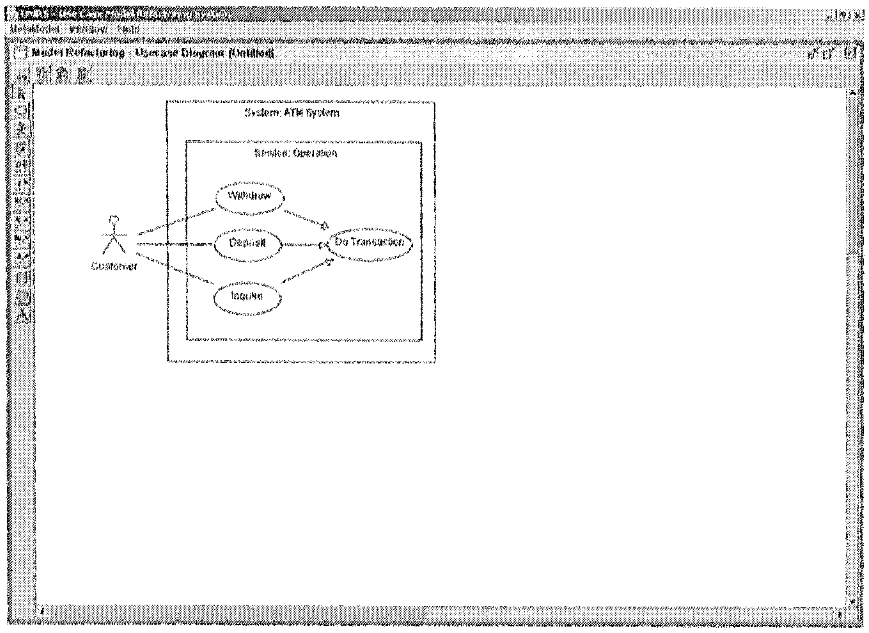


Figure 30. Use case model of ATM system after Step 5

6. Because of security reasons, every customer's operations should be kept in his own session. In the session, customer will be asked to input his PIN to validate his identity. The ATM system will do approval process to check the PIN. If the PIN is not correct, invalid PIN extension will be processed from within a transaction.

   1) Use 'Create Empty Use Case' to create use case 'Open Session', 'Do Approval Process' and 'Process Invalid PIN'
   2) Create Association between actor 'Customer' and use case 'Open Session'.
   3) Use 'Create Empty Actor' to create actor 'Bank'
   4) Create Association between actor 'Bank' and use case 'Do Transaction'
   5) Remove the Associations between actor 'Customer' and use cases 'Withdraw', 'Deposit' and 'Inquire'

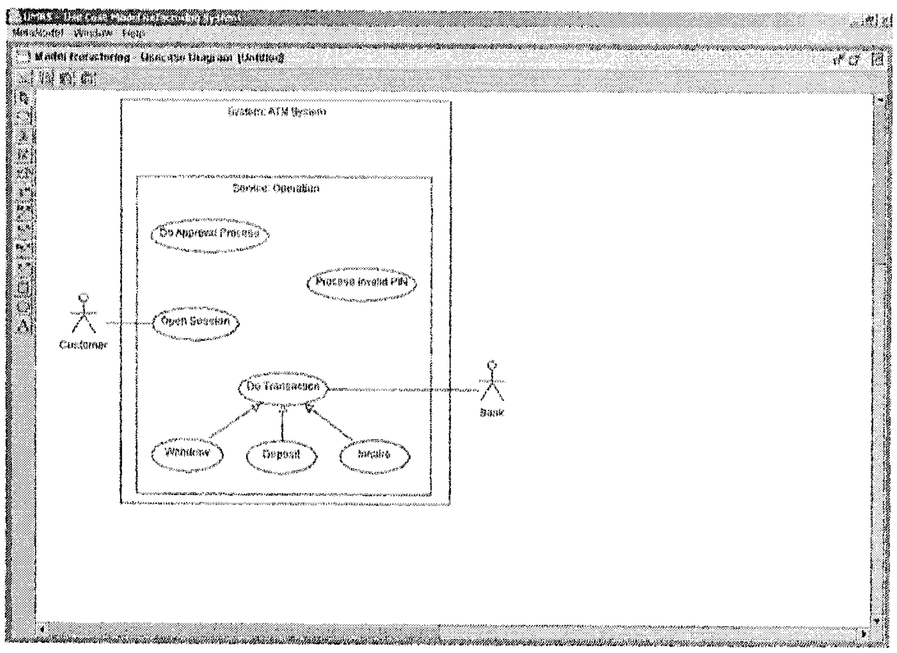
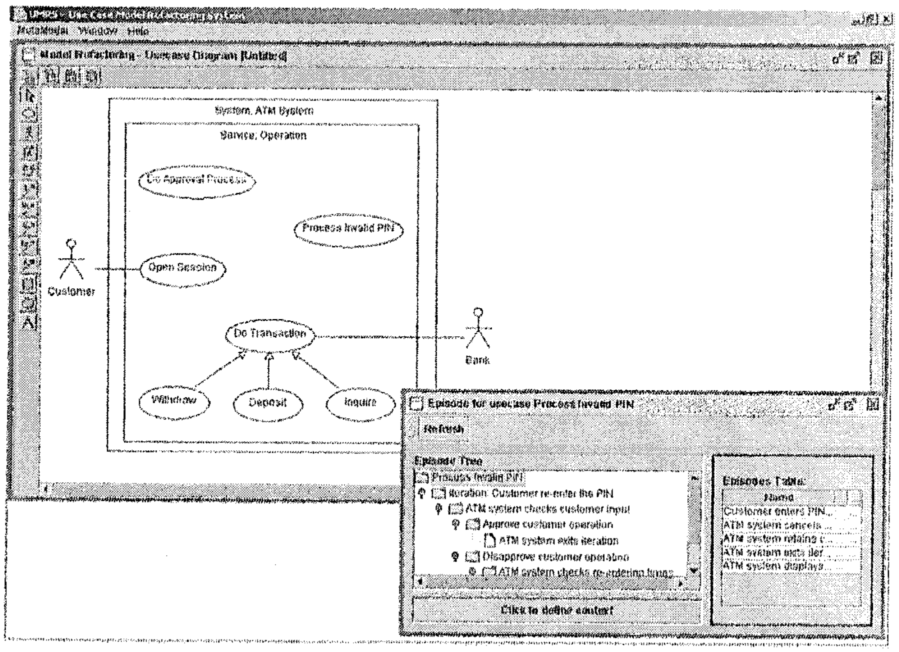


Figure 31. Use case model of ATM system after Step 6

7. Define episodes, pre-conditions, context and post-conditions of new use cases 'Open Session', 'Do Approval Process' and 'Process Invalid PIN'



Process Invalid PIN
Iteration: Customer re-enter the PIN
ATM system checks customer input
Approve customer operation
ATM system exits iteration
Disapprove customer operation
ATM system checks re-entering times...

Name
Customer enters PIN...
ATM system cancels ...
ATM system retains c...
ATM system adds iter...
ATM system displays...

Figure 32. Use case model of ATM system after Step 7

8. Make use case 'Open Session' include use case 'Do Approval Process' and 'Do Transaction'. Add use case 'Process Invalid PIN' as an extension of use case 'Do Transaction'

    1) Use 'Include Use Case' to make use case 'Open Session' include use case 'Do Transaction'

    2) Use 'Include Use Case' to make use case 'Open Session' include use case 'Do Approval process'

    3) Create 'Extend' between use case 'Process Invalid PIN' and 'Do Transaction'

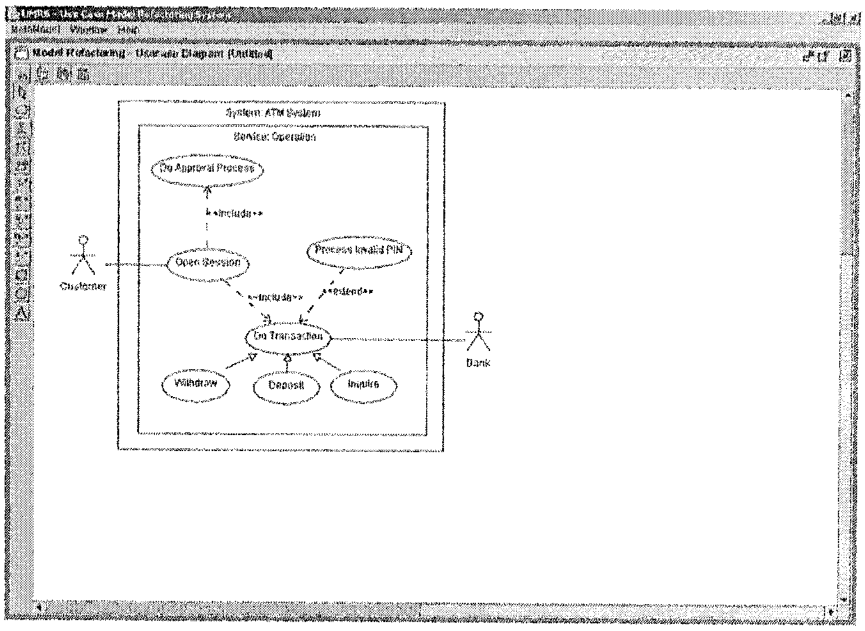


Figure 33. Use case model of ATM system after Step 8

9. Customers need to transfer funds between different accounts. A new use case 'Transfer' is created. It should support transaction. Its episodes, pre-conditions, context, post-conditions and events are defined as well.

    1) Use 'Create Empty Use Case' to create use case 'Transfer'

2) Define pre-conditions, context and post-condition of use case 'Transfer'

3) Define episodes of use case 'Transfer'. (Some episodes are imported from use case 'Transaction')

4) Define events of episodes in use case 'Transfer'

5) Use 'Change Parent Use Case' to make use case 'Do Transaction' as the parent of use case 'Transfer'

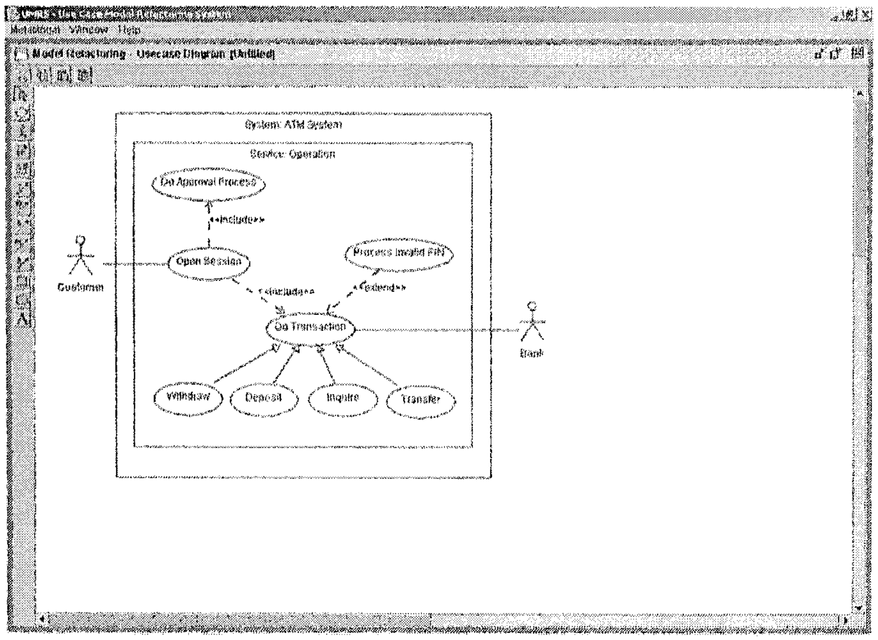


Figure 34. Use case model of ATM system after Step 9

10. ATM system needs to be maintained by the operator from bank. New use cases 'Start System' and 'Stop System' are created. The new use cases are grouped into a new service. The episodes, pre-conditions, context, post-conditions and events of new use cases are defined. A new actor 'Operator' is added to interact with the new use cases.

1) Use 'Create Empty Use Case' to create use case 'Start System' and 'Stop System '

2) Define episodes, preconditions, context and post-condition of use case 'Start

System' and 'Stop System'

3) Define events of episodes in use case 'Start System' and 'Stop System'

4) Use 'Create Empty Actor' to create actor 'Operator'

5) Define goals of actor 'Operator'

6) Create associations between actor 'Operator' and use cases 'Start System', 'Stop System'

7) Use 'Create Empty Service' to create service 'Maintenance'

8) Put use cases 'Start System' and 'Stop System' in the service 'Maintenance'

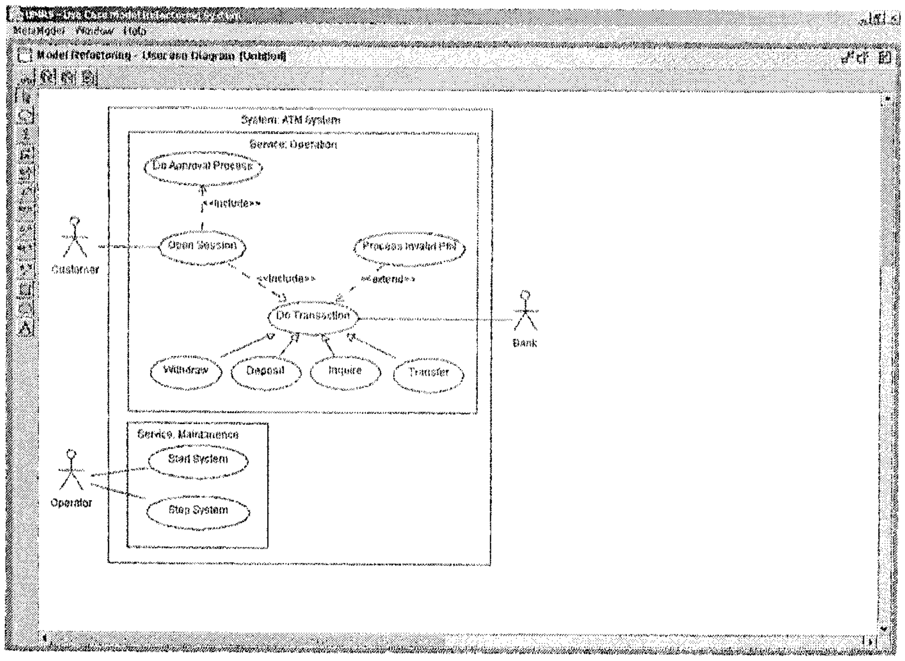


Figure 35. Use case model of ATM system after Step 10

6.3 Summary

After ten steps, an ATM use case model is defined according to the requirements. By inspecting the process of creating a metamodel, I get a conclusion of necessary steps of making a use case metamodel with the refactoring tool. The steps are:

1) analyze the requirements and get a brief description
2) define use cases and actors
3) define the episodes of use cases and goals of actors
4) define context, precondition and post-condition of use case
5) define the events of the episode
6) apply refactoring rules to the model according to the changes on requirements
7) repeat step 2) – step 6) until the model meeting the requirements

This case study shows all the aspects of getting a use case metamodel by using the tool. It proves the feasibility of use case model and the refactorings on the model.

7. Conclusions

This thesis introduces the practice in designing and implementing a refactoring tool for defining our use case metamodel and applying refactorings to the model. A use case metamodel and some refactoring rules are introduced and implemented in the refactoring tool. The tool has facilitated generating use case model greatly. The works I have done are a) describe refactoring rules related to use case and actor in detail with arguments, pre-conditions and post-conditions b) define data structure of use case model on environment level d) extend Drawlets framework to be the base framework of the tool e) design and implement the use case model editor of environment level f) implement some of the refactoring rules related to use case and actor g) make a case study with the tool. The works haven't been finished are a) refactoring rules in move category, distribute category and some rules in compose category are not implemented b) extension of interfaces and classes in Drawlets framework for structure and event level are not provided. In the case study, a use case metamodel of ATM system is created with the refactoring tool. The model is improved by applying refactoring rules and some other extension functions provided by the refactoring tool to the model according to the changes on requirements. The result shows that making refactoring on use case model is feasible. Refactorings facilitate the process of reusing requirement fragments, which reduces elaboration time and improves requirement quality.

The other two students' work also shows some issues on the implementation such as a) the representation of structure and event level are not straightforward b) not all the refactoring rules are implemented, for example, the refactoring on the event level c) goal model are not fully defined and implemented d) some refactoring rules need to be improved for better description, validation and implementation, etc.

In the future, following features may be added to the refactoring tool for facilitating the refactoring process and making use case refactorings more practical.

1. scripting

   Allow users to write scripts for metamodel definition and refactoring

2. redo and undo

   Allow users to recover to the previous status or reapply the refactorings

3. data exchange

   Help users to exchange data with other modeling systems. The issue could be faced is that other use case modeling tools may not have the correspondent levels defined in our metamodel or vice versa. The information of environment level may be exchanged without big changes while the other two levels may not be exchanged directly. Users may have to define the structure level and event level by themselves after the use case model imported.

4. model optimization

Help users to analyze metamodel according to pre-defined use case design patterns. Some refactoring suggestions of how to optimizing model and possible results after optimization are shown to users. The tool could finish the refactoring process automatically if users satisfy with the result of refactoring on the model.

5. more graphical representations

   Consider using graphical representations on structure level and event level. Currently tree structure is used on these two levels. When composite episode or event is defined in the model, the tree structure may not show the sequences of the atomic episode or event clearly. With graphical representations, users can understand the definition of the composite episode or event easily. In order to use more graphical representation, extensions of interfaces and objects of the Drawlets framework should be provided.

Refactoring rules may be added or altered according to the feedbacks from the users of the tool. Test model may be set up for verifying that behaviors of use case model are preserved after refactoring applied. More case studies are needed to help on evaluating the metamodel and refactoring rules, and improving the refactoring tool as well.

References

1. Jay Banerjee and Won Kim, "Semantics and implementation of schema evolution in object-oriented databases", In Proceedings of the ACM SIGMOD Conference, 1987.
2. Russell C. Bjork, "ATM Simulation", Gordon College, 2002. http://www.math-cs.gordon.edu/local/courses/cs211/ATMExample/
3. Ray Buhr, "Use case maps (UCMs) Updated: A Simple Visual Notatieeon for Understanding and Architecting the Emergent Bezhavior of Large, Complex, Self Modifying Systems", 1995. ftp://ftp.sce.carleton.ca/pub/UseCaseMaps/ucmUpdate.ps
4. Gregory Butler and Lugang Xu, "Cascaded refactoring for framework evolution", In Proceedings of 2001 Symposium on Software Reusability, ACM Press, pp.51-57, 2001.
5. Alistair Cockburn, "Structuring Use Cases with Goals" – JOOP/ROAD Vol. 10(5), pp.35-40, Sep.1997 and Vol. 10(7), pp.56-62, Nov.1997.
6. Larry L. Constantine and Lucy A. D. Lockwood, "Structure and style in use cases for user interface design", Addison-Wesley, ISBN:0-201-65789-9, pp.245-279, 2001.
7. A. Dardenne, A. van Lamsweerde and S. Fickas, "Goal-directed Requirements Acquisition", Science of Computer Programming, Vol. 20, pp.3-50, 1993.

8. D. Firesmith, B. Henderson-Sellers and I. Graham, "OPEN Modeling Language (OML) Reference Manual". Sigs, New York, 1997.

9. M. Fowler, "Refactoring: Improving the Design of Existing Code", Addison-Wesley, ISBN:0-201-48567-2, 1999.

10. Erich Gamma, Richard Helm, Ralph Johnson, and John Viissides, "Design Patterns - Elements of Reusable Object-Oriented Software", Addison-Wesley, Reading, Massachusetts, 1995.

11. I. Graham, "Migrating to Object Technology", Addison-Wesley, ISBN:0-201-59389-0, 1995.

12. B. Henderson-Sellers and D.G. Firesmith, "Choosing between OPEN and UML", American Programmer, 10(3), pp.15-23 COTAR Contribution no 97/5, 1997.

13. Brian Henderson-Sellers, Donald G. Firesmith, and Ian Graham, "OML Metamodel: Relationships and state modeling", JOOP Vol. 10(1), pp.47-51, March/April 1997.

14. I. Jacobson, G. Booch and J. Rumbaugh, "The Unified Software Development Process". Addison Wesley Longman, ISBN:0-201-57169-2, 1999.

15. I. Jacobson, M. Christerson, P. Jonsson and G. Overgaard, "Object-Oriented Software Engineering, A Use Case Driven Approach", Addison-Wesley, Wokingham, ISBN:O-201-54435-0, 1992.

16. Ivar Jacobson, Maria Ericsson and Agneta Jacobson, "the Object Advantage – Business Process Reengineering with Object Technology", ACM Press, ISBN:O-201-42289-1, 1994.

17. W. F. Opdyke, "Refactoring Object-Oriented Frameworks", Ph.D. thesis, University of Illinois, 1992.

18. C. Potts, K. Takahashi, A. Anton, "Inquiry-Based Requirements Analysis", IEEE Software, pp.21-32, March 1994.

19. Rational, UML Summary, Semantics, Notation Guide, Version 1.1, Rational Software Corporation, 1997.

20. Bjorn Regnell, "Requirements Engineering with Use Cases -- a Basis for Software Development". Ph.D. thesis, Lund University, 1999.

21. Donald Bradley Roberts. "Practical Analysis for Refactoring", Ph.D. thesis, University of Illinois, 1999.

22. RoleModel Software, "Drawlets Framework", 2003. http://www.rolemodelsoftware.com/drawlets/index.php

23. Kexing Rui, "Refactoring Use Case Models", Thesis proposal, University of Concordia, 2002.

24. Kexing Rui, Shengbing Ren, Gregory Butler, "Refactoring Use Case Models: A Case Study", University of Concordia, 2002.

25. J. Rumbaugh, I. Jacobson and G. Booch, "The Unified Modeling Language Reference Manual", Reading, MA: Addison-Wesley, ISBN:0-201-30998-X, 1999.

26. Hossein Saiedian, "Introduction to Requirement Engineering Management", University of Kansas, 2003.

27. Lance Tokuda, "Evolving Object-Oriented Designs with Refactorings", Ph.D. thesis,

University of Texas at Austin, 1999.

28. Y. Yamane, N. Igata and I. Namba, "High-performance XML Storage/Retrieval System", FUJITSU Sci. Tech. J., 36(2), pp.185-192, 2000.

29. Didar Zowghi, "RE@UTS - Requirements Engineering Activities", 2002. http://research.it.uts.edu.au/re/re_uts_activities.html

Appendices

A. Architecture of Refactoring Tool

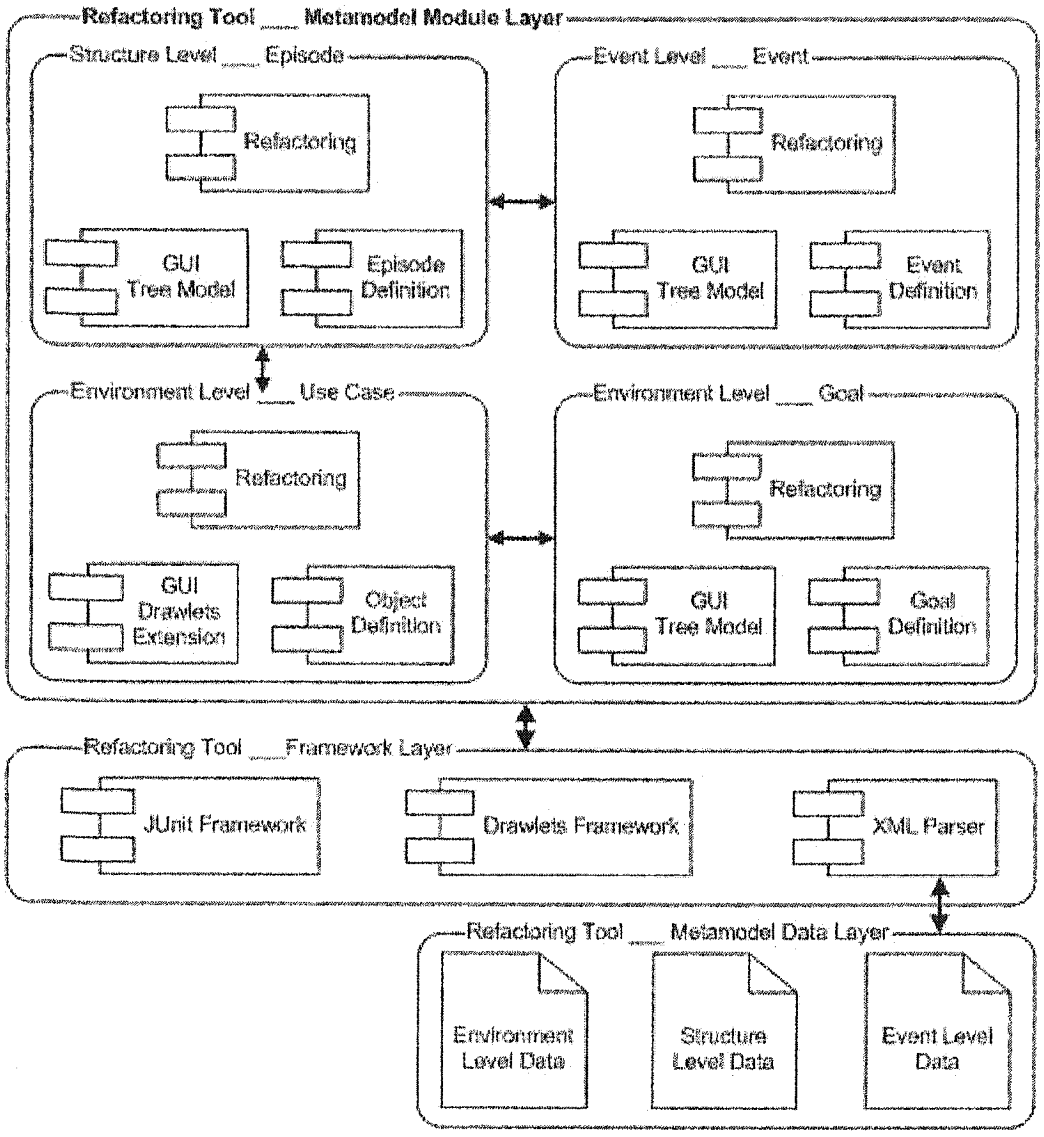


Figure 36. Architecture of Refactoring Tool

B. Use Case Metamodel of ATM system

B.1 Environment Level

```
<UsecaseModel>
    <Usecases>
        <Usecase id="60">
            <Name>Do Transaction</Name>
            <Coordinate x="257" y="229"/>
            <Size width="105" height="58"/>
            <Description></Description>
            <EpisodeID>60</EpisodeID>
        </Usecase>
        <Usecase id="91">
            <Name>Process Invalid PIN</Name>
            <Coordinate x="355" y="149"/>
            <Size width="100" height="58"/>
            <Description></Description>
            <EpisodeID>91</EpisodeID>
        </Usecase>
        <Usecase id="41">
            <Name>Withdraw</Name>
            <Coordinate x="123" y="293"/>
            <Size width="101" height="58"/>
            <Description>Withdraw money from the given bank account</Description>
            <EpisodeID>41</EpisodeID>
        </Usecase>
        <Usecase id="42">
            <Name>Deposit</Name>
            <Coordinate x="215" y="293"/>
            <Size width="100" height="58"/>
            <Description>Deposit money to the selected bank acount</Description>
            <EpisodeID>42</EpisodeID>
        </Usecase>
        <Usecase id="43">
            <Name>Inquire</Name>
            <Coordinate x="306" y="292"/>
            <Size width="100" height="58"/>
            <Description>Get account information from the selected account.</Description>
            <EpisodeID>43</EpisodeID>
        </Usecase>
        <Usecase id="89">
            <Name>Open Session</Name>
            <Coordinate x="132" y="149"/>
            <Size width="100" height="58"/>
            <Description></Description>
            <EpisodeID>89</EpisodeID>
        </Usecase>
        <Usecase id="90">
            <Name>Do Approval Process</Name>
            <Coordinate x="115" y="58"/>
            <Size width="135" height="58"/>
            <Description></Description>
            <EpisodeID>90</EpisodeID>
        </Usecase>
        <Usecase id="99">
            <Name>Transfer</Name>
            <Coordinate x="401" y="294"/>
            <Size width="100" height="59"/>
            <Description></Description>
            <EpisodeID>99</EpisodeID>
        </Usecase>
        <Usecase id="138">
            <Name>Start System</Name>
            <Coordinate x="120" y="395"/>
            <Size width="135" height="58"/>
            <Description></Description>
            <EpisodeID>138</EpisodeID>
        </Usecase>
        <Usecase id="139">
            <Name>Stop System</Name>
            <Coordinate x="118" y="453"/>
            <Size width="139" height="58"/>
            <Description></Description>
            <EpisodeID>139</EpisodeID>
        </Usecase>
    </Usecases>
    <Actors>
        <Actor id="40">
```

```
            <Name>Customer</Name>
            <Coordinate x="8" y="138"/>
            <Size width="75" height="88"/>
            <Description>Normally, Customers have an account opened at the bank. They can
access to their account to do some transcations like withdrawing money, etc.</Description>
            <UserID>40</UserID>
        </Actor>
        <Actor id="92">
            <Name>Bank</Name>
            <Coordinate x="536" y="220"/>
            <Size width="50" height="88"/>
            <Description></Description>
            <UserID>92</UserID>
        </Actor>
        <Actor id="141">
            <Name>Operator</Name>
            <Coordinate x="7" y="406"/>
            <Size width="68" height="88"/>
            <Description></Description>
            <UserID>141</UserID>
        </Actor>
    </Actors>
    <Relationship>
        <Association>
            <Actor-Usecase>
                <ActorRef>92</ActorRef>
                <UsecaseRef>60</UsecaseRef>
            </Actor-Usecase>
            <Actor-Usecase>
                <ActorRef>40</ActorRef>
                <UsecaseRef>89</UsecaseRef>
            </Actor-Usecase>
            <Actor-Usecase>
                <ActorRef>141</ActorRef>
                <UsecaseRef>138</UsecaseRef>
            </Actor-Usecase>
            <Actor-Usecase>
                <ActorRef>141</ActorRef>
                <UsecaseRef>139</UsecaseRef>
            </Actor-Usecase>
        </Association>
        <Generalization>
            <Usecase-Usecase>
                <UsecaseRef1>43</UsecaseRef1>
                <UsecaseRef2>60</UsecaseRef2>
            </Usecase-Usecase>
            <Usecase-Usecase>
                <UsecaseRef1>42</UsecaseRef1>
                <UsecaseRef2>60</UsecaseRef2>
            </Usecase-Usecase>
            <Usecase-Usecase>
                <UsecaseRef1>41</UsecaseRef1>
                <UsecaseRef2>60</UsecaseRef2>
            </Usecase-Usecase>
            <Usecase-Usecase>
                <UsecaseRef1>99</UsecaseRef1>
                <UsecaseRef2>60</UsecaseRef2>
            </Usecase-Usecase>
        </Generalization>
        <Inclusion>
            <Usecase-Usecase>
                <UsecaseRef1>89</UsecaseRef1>
                <UsecaseRef2>60</UsecaseRef2>
            </Usecase-Usecase>
            <Usecase-Usecase>
                <UsecaseRef1>89</UsecaseRef1>
                <UsecaseRef2>90</UsecaseRef2>
            </Usecase-Usecase>
        </Inclusion>
        <Extension>
            <Usecase-Usecase>
                <UsecaseRef1>91</UsecaseRef1>
                <UsecaseRef2>60</UsecaseRef2>
            </Usecase-Usecase>
        </Extension>
    </Relationship>
    <Other>
       <ServiceBound id="140">
            <Name>Maintanenence</Name>
            <BoundRect x="106" y="379" width="161" height="142"/>
            <Description></Description>
            <UsecaseEID></UsecaseEID>
        </ServiceBound>
        <ServiceBound id="55">
            <Name>Operation</Name>
            <BoundRect x="107" y="45" width="399" height="321"/>
            <Description></Description>
            <UsecaseEID></UsecaseEID>
        </ServiceBound>
        <SystemBound>
            <Name>ATM System</Name>
            <BoundRect x="84" y="17" width="438" height="522"/>
            <Description></Description>
        </SystemBound>
    </Other>
</UsecaseModel>
```

<GoalModel>

```
    <Goal name="Goal: Withdraw" description="Achieve getting money from ATM machine">
        text
        <Actor name="All" description="">text</Actor>
    </Goal>
    <Goal name="Goal: Deposit" description="Achieve depositing money to given account">
        text
        <Actor name="All" description="">text</Actor>
    </Goal>
    <Goal name="Goal: Inquire" description="Achieve getting account information">
        text
        <Actor name="All" description="">text</Actor>
    </Goal>
```

</GoalModel>

B.2 Structure Level

<EpisodeModel>

<UseCase ID="41" PreCondition="1. Customer should have a valid bank card which is accepted by the ATM machine 2. Customer should know the PIN of the card as well. 3. ATM is working properly." Context=" A withdrawal transaction asks the customer to choose a type of account to withdraw from (e.g. checking) from a menu of possible accounts, and to choose a dollar amount from a menu of possible amounts. The system verifies that it has sufficient money on hand to satisfy the request before sending the transaction to the bank. (If not, the customer is informed and asked to enter a different amount.) If the transaction is approved by the bank, the appropriate amount of cash is

```
dispensed by the machine before it issues a receipt. (The dispensing of cash is also recorded in the
ATM's log.)    A withdrawal transaction can be cancelled by the customer pressing the Cancel key
any time prior to choosing the dollar amount. " PostCondition="1. ATM machine will return to the
login   screen for next customer. 2. Customers will get the amount of money they request.">
<Episode name="41" type="Sequence Episode">
  <SubEpisodes>
    <SubEpisode>Customer inserts bank card</SubEpisode>
    <SubEpisode>Customer enters PIN code</SubEpisode>
    <SubEpisode>Customer selects bank account</SubEpisode>
    <SubEpisode>Customer enters amount of money</SubEpisode>
    <SubEpisode>ATM system does withdrawal processing</SubEpisode>
    <SubEpisode>ATM machine ejects bank card</SubEpisode>
  </SubEpisodes>
</Episode>
<Episode name="ATM System does withdrawal processing" type="Alternation Episode">
  <SubEpisodes>
    <SubEpisode>Bank allows operation</SubEpisode>
    <SubEpisode>Bank denies operation</SubEpisode>
  </SubEpisodes>
</Episode>
<Episode name="Bank allows operation" type="Sequence Episode">
  <SubEpisodes>
    <SubEpisode>Customer gets cash</SubEpisode>
    <SubEpisode>ATM system updates bank account balance</SubEpisode>
  </SubEpisodes>
</Episode>
<Episode name="Customer enters amount of money" type="Atomic Episode" />
<Episode name="Customer gets cash" type="Atomic Episode" />
<Episode name="ATM system updates bank account balance" type="Atomic Episode" />
</UseCase>
<UseCase ID="42" PreCondition="1. Customer should have a valid   bank card which is accepted by
the ATM machine 2. Customer should know the PIN of the card as well. 3. ATM is working properly."
Context=" A deposit transaction asks the customer to choose a type of account to deposit to (e.g.
checking) from a menu of possible accounts, and to type in a dollar amount on the keyboard. The
transaction is initially sent to the bank to verify that the ATM can accept a deposit from this customer
to this account. If the transaction is approved, the machine accepts an envelope from the customer
containing cash and/or checks before it issues a receipt. Once the envelope has been received, a
second message is sent to the bank, to confirm that the bank can credit the customer's account -
contingent on manual verification of the deposit envelope contents by an operator later. (The receipt of
an envelope is also recorded in the ATM's log.)  A deposit transaction can be cancelled by the
customer pressing the Cancel key any time prior to inserting the envelope   containing the deposit.
```

```
The transaction is automatically cancelled if the customer fails to insert the envelope containing the
deposit within a reasonable period of time after being asked to do so. " PostCondition="1. ATM
machine will return to the login screen for next customer. 2. Customers will receive a receipt.">
<Episode name="42" type="Sequence Episode">
  <SubEpisodes>
    <SubEpisode>Customer inserts bank card</SubEpisode>
    <SubEpisode>Customer enters PIN code</SubEpisode>
    <SubEpisode>Customer selects bank account</SubEpisode>
    <SubEpisode>Customer enters amount of money</SubEpisode>
    <SubEpisode>ATM system does deposit processing</SubEpisode>
    <SubEpisode>ATM machine ejects bank card</SubEpisode>
  </SubEpisodes>
</Episode>
<Episode name="ATM System does deposit processing" type="Alternation Episode">
  <SubEpisodes>
    <SubEpisode>Bank allows operation</SubEpisode>
    <SubEpisode>Bank denies operation</SubEpisode>
  </SubEpisodes>
</Episode>
<Episode name="Bank allows operation" type="Sequence Episode">
  <SubEpisodes>
    <SubEpisode>Customer inserts envelope</SubEpisode>
    <SubEpisode>ATM system updates bank account balance</SubEpisode>
    <SubEpisode>Customer gets receipt</SubEpisode>
  </SubEpisodes>
</Episode>
<Episode name="ATM system updates bank account balance" type="Atomic Episode" />
<Episode name="Customer inserts envelope" type="Atomic Episode" />
<Episode name="Customer enters amount of money" type="Atomic Episode" />
<Episode name="Customer gets receipt" type="Atomic Episode" />
</UseCase>
<UseCase ID="43" PreCondition="1. Customers should have a valid bank card which is accepted by
the ATM machine  2. Customers should know the PIN of the card as well.  3. Customers should
have a bank book if they want to  update the information on it." Context="  An inquiry transaction
asks the customer to choose a type of account to inquire about from a menu of possible accounts. No
further action is required once the  transaction is approved by the bank before printing the receipt.
An inquiry transaction can be cancelled by the customer pressing the Cancel key any time prior to
choosing the account to inquire about.  " PostCondition="1. ATM machine will return to the login
screen for next customer. 2. Bank book may be updated.">
<Episode name="43" type="Sequence Episode">
  <SubEpisodes>
```

```
      <SubEpisode>Customer inserts bank card</SubEpisode>
      <SubEpisode>Customer enters PIN code</SubEpisode>
      <SubEpisode>Customer selects bank account</SubEpisode>
      <SubEpisode>ATM system does inquiry processing</SubEpisode>
      <SubEpisode>ATM machine ejects bank card</SubEpisode>
   </SubEpisodes>
</Episode>
<Episode name="ATM system does inquiry processing" type="Alternation Episode">
   <SubEpisodes>
      <SubEpisode>Bank allows operation</SubEpisode>
      <SubEpisode>Bank denies operation</SubEpisode>
   </SubEpisodes>
</Episode>
<Episode name="Bank allows operation" type="Alternation Episode">
   <SubEpisodes>
      <SubEpisode>ATM system prints account information on screen</SubEpisode>
      <SubEpisode>ATM system prints account information on receipt</SubEpisode>
      <SubEpisode>ATM system updates bankbook</SubEpisode>
   </SubEpisodes>
</Episode>
<Episode name="ATM system prints account information on screen" type="Atomic Episode" />
<Episode name="ATM system prints account information on receipt" type="Atomic Episode" />
<Episode name="ATM system updates bankbook" type="Atomic Episode" />
</UseCase>
<UseCase ID="60" PreCondition=" " Context=" " PostCondition=" ">
<Episode name="60" type="Sequence Episode" />
<Episode name="Customer selects bank account" type="Atomic Episode" />
<Episode name="Customer inserts bank card" type="Atomic Episode" />
<Episode name="Customer enters PIN code" type="Atomic Episode" />
<Episode name="ATM machine ejects bank card" type="Atomic Episode" />
<Episode name="Bank denies operation" type="Atomic Episode" />
</UseCase>
<UseCase ID="89" PreCondition="1. Bank card should be inserted into the ATM machine" Context="
A session is started when a customer inserts an ATM card into the card reader slot of the machine. The
ATM pulls the card into the machine and reads it. (If the reader cannot read the card due to improper
insertion or a damaged stripe, the card is ejected, an error screen is displayed, and the session is
aborted.) The customer is asked to enter his/her PIN, and is then allowed to perform one or more
transactions, choosing from a menu of possible types of transaction in each case. After each
transaction, the customer is asked whether he/she would like to perform another. When the customer is
through performing transactions, the card is ejected from the machine and the session ends. If a
transaction is aborted due to too many invalid PIN entries, the session is also aborted, with the card
```

```
being retained in the machine.  The customer may abort the session by pressing the Cancel key when
entering a PIN or choosing a transaction type. " PostCondition="1. Bank cards should be ejected. 2.
Money may be dispensed upon the customers' requests 3. Login screen is shown on the ATM
machine">
<Episode name="89" type="Sequence Episode" />
<Episode name="Customer selects bank account" type="Atomic Episode" />
<Episode name="Customer inserts bank card" type="Atomic Episode" />
<Episode name="Customer enters PIN code" type="Atomic Episode" />
<Episode name="ATM machine ejects bank card" type="Atomic Episode" />
<Episode name="ATM system sends PIN to bank" type="Atomic Episode" />
<Episode name="ATM system gets result from bank" type="Atomic Episode" />
</UseCase>
<UseCase ID="91" PreCondition="the bank reports that the customer's transaction is disapproved due
to an invalid PIN" Context=" An invalid PIN extension is started from within a transaction when the
bank reports that the customer's transaction is disapproved due to an invalid PIN. The customer is
required to re-enter the PIN and the original request is sent to the bank again. If the bank now
approves the transaction, or disapproves it for some other reason, the original use case is continued;
otherwise the process of re-entering the PIN is repeated. Once the PIN is successfully re-entered, it is
used for both the current transaction and all subsequent transactions in the session. If the customer
fails three times to enter the correct PIN, the card is permanently retained, a screen is displayed
informing the customer of this and suggesting he/she contact the bank, and the entire customer session
is aborted.  If the customer presses Cancel instead of re-entering a PIN, the original transaction is
cancelled. " PostCondition="If the bank now approves the transaction, or disapproves it for some other
reason, the original use case is continued; otherwise the process of re-entering the PIN is repeated.">
<Episode name="91" type="Sequence Episode">
  <SubEpisodes>
    <SubEpisode>Iteration: Customer re-enters the PIN</SubEpisode>
  </SubEpisodes>
</Episode>
<Episode name="Iteration: Customer re-enters the PIN" type="Iteration Episode">
  <SubEpisodes>
    <SubEpisode>ATM system checks customer input</SubEpisode>
  </SubEpisodes>
</Episode>
<Episode name="ATM system checks customer input" type="Alternation Episode">
  <SubEpisodes>
    <SubEpisode>Approve customer operation</SubEpisode>
    <SubEpisode>Disapprove customer operation</SubEpisode>
  </SubEpisodes>
</Episode>
<Episode name="Approve customer operation" type="Sequence Episode">
```

```
  <SubEpisodes>
    <SubEpisode>ATM system exits iteration</SubEpisode>
  </SubEpisodes>
</Episode>
<Episode name="Disapprove customer operation" type="Sequence Episode">
  <SubEpisodes>
    <SubEpisode>ATM system checks re-entering times</SubEpisode>
  </SubEpisodes>
</Episode>
<Episode name="ATM system checks re-entering times" type="Alternation Episode">
  <SubEpisodes>
    <SubEpisode>Customer re-enters over 3 times</SubEpisode>
    <SubEpisode>Customer re-enters less than 3 times</SubEpisode>
  </SubEpisodes>
</Episode>
<Episode name="Customer re-enters over 3 times" type="Alternation Episode">
  <SubEpisodes>
    <SubEpisode>ATM system retains card</SubEpisode>
    <SubEpisode>ATM system displays error</SubEpisode>
  </SubEpisodes>
</Episode>
<Episode name="Customer re-enters less than 3 times" type="Alternation Episode">
  <SubEpisodes>
    <SubEpisode>Customer enters PIN code</SubEpisode>
  </SubEpisodes>
</Episode>
<Episode name="Customer enters PIN code" type="Atomic Episode" />
<Episode name="ATM system cancels operation" type="Atomic Episode" />
<Episode name="ATM system retains card" type="Atomic Episode" />
<Episode name="ATM system exits iteration" type="Atomic Episode" />
<Episode name="ATM system displays error" type="Atomic Episode" />
</UseCase>
<UseCase ID="90" PreCondition="PIN has been entered or re-entered. It's going to be sent to the bank for validation" Context="A approval process is started when PIN has been entered. PIN will be validated at bank. If it's approved, customers can do transactions with bank. Otherwise, customers will be asked for re-entering the PIN." PostCondition="Customer should get the approval or disapproval of doing furthur transactions. If approved, transaction menu will be shown on the screen. If not approved, entering PIN screen will be shown instead.">
<Episode name="90" type="Sequence Episode">
  <SubEpisodes>
    <SubEpisode>ATM system sends PIN to bank</SubEpisode>
    <SubEpisode>ATM system gets result from bank</SubEpisode>
  </SubEpisodes>
</Episode>
<Episode name="ATM system sends PIN to bank" type="Atomic Episode" />
<Episode name="ATM system gets result from bank" type="Atomic Episode" />
</UseCase>
<UseCase ID="99" PreCondition="Customer should have more than one type of account." Context="
A transfer transaction asks the customer to choose a type of account to transfer from (e.g. checking)
from a menu of possible accounts, to choose a different account to transfer to, and to type in a dollar
amount on the keyboard. No further action is required once the transaction is approved by the bank
before printing the receipt.  A transfer transaction can be cancelled by the customer pressing the
Cancel key any time prior to entering a dollar amount. " PostCondition="Money has been transfered
and exchange rate is applied if needed">
<Episode name="99" type="Sequence Episode">
  <SubEpisodes>
    <SubEpisode>Customer inserts bank card</SubEpisode>
    <SubEpisode>Customer enters PIN code</SubEpisode>
    <SubEpisode>Customer selects 'From which account'</SubEpisode>
    <SubEpisode>Customer selects 'To which account'</SubEpisode>
    <SubEpisode>Customer enters transfer amount</SubEpisode>
    <SubEpisode>ATM system does transfer processing</SubEpisode>
    <SubEpisode>ATM machine ejects bank card</SubEpisode>
  </SubEpisodes>
</Episode>
<Episode name="ATM system does transfer processing" type="Alternation Episode">
  <SubEpisodes>
    <SubEpisode>Bank allows operation</SubEpisode>
    <SubEpisode>Bank denies operation</SubEpisode>
  </SubEpisodes>
</Episode>
<Episode name="Bank allows operation" type="Sequence Episode">
  <SubEpisodes>
    <SubEpisode>Customer gets receipt</SubEpisode>
  </SubEpisodes>
</Episode>
<Episode name="Customer selects 'From which account'" type="Atomic Episode" />
<Episode name="Customer selects 'To which account'" type="Atomic Episode" />
<Episode name="Customer gets receipt" type="Atomic Episode" />
<Episode name="Customer enters transfer amount" type="Atomic Episode" />
</UseCase>
<UseCase ID="138" PreCondition="Operator should know the user name and password." Context="
```

```
The system is started up when the operator turns the operator switch to the &quot;on&quot; position.
The operator will be asked to enter the amount of money currently in the cash dispenser, and a
connection to the bank will be established. Then the servicing of customers can begin. "
PostCondition=" ATM machine is ready for using. Some money has been put in the dispenser. The
connection to the bank has been established.">
<Episode name="138" type="Sequence Episode">
  <SubEpisodes>
    <SubEpisode>Operator turns on the ATM machine</SubEpisode>
    <SubEpisode>ATM System starts</SubEpisode>
  </SubEpisodes>
</Episode>
<Episode name="ATM System starts" type="Alternation Episode">
  <SubEpisodes>
    <SubEpisode>ATM machine starts successfully</SubEpisode>
    <SubEpisode>ATM machine couldn't be started</SubEpisode>
  </SubEpisodes>
</Episode>
<Episode name="ATM machine starts successfully" type="Sequence Episode">
  <SubEpisodes>
    <SubEpisode>Operator inputs administrator password</SubEpisode>
    <SubEpisode>ATM system does password checking</SubEpisode>
  </SubEpisodes>
</Episode>
<Episode name="ATM system does password checking" type="Alternation Episode">
  <SubEpisodes>
    <SubEpisode>Operator is allowed to enter the system</SubEpisode>
    <SubEpisode>Operator is not allowed to enter the system</SubEpisode>
  </SubEpisodes>
</Episode>
<Episode name="Operator is allowed to enter the system" type="Sequence Episode">
  <SubEpisodes>
    <SubEpisode>Operator enters the amount of money currently in the dispenser</SubEpisode>
    <SubEpisode>ATM system establishes the connection to the bank</SubEpisode>
  </SubEpisodes>
</Episode>
<Episode name="Operator turns on the ATM machine" type="Atomic Episode" />
<Episode name="Operator inputs administrator password" type="Atomic Episode" />
<Episode name="Operator enters the amount of money currently in the dispenser" type="Atomic
Episode" />
<Episode name="ATM system establishes the connection to the bank" type="Atomic Episode" />
<Episode name="ATM machine couldn't be started" type="Atomic Episode" />
```

```
<Episode name="Operator is not allowed to enter the system" type="Atomic Episode" />
</UseCase>
<UseCase ID="139" PreCondition="No customer is using the ATM machine" Context="   The system
is shut down when the operator makes sure that no customer is using the machine, and then turns the
operator switch to the &quot;off&quot; position. The connection to the bank will be shut down. Then
the operator is free to remove deposited envelopes, replenish cash and paper, etc. " PostCondition="
ATM machine is turned off and the connection to the bank is cut down.">
<Episode name="139" type="Sequence Episode">
  <SubEpisodes>
    <SubEpisode>Operator inputs administrator password</SubEpisode>
    <SubEpisode>ATM system does password checking</SubEpisode>
  </SubEpisodes>
</Episode>
<Episode name="ATM system does password checking" type="Alternation Episode">
  <SubEpisodes>
    <SubEpisode>Operator is allowed to enter the system</SubEpisode>
    <SubEpisode>Operator is not allowed to enter the system</SubEpisode>
  </SubEpisodes>
</Episode>
<Episode name="Operator is allowed to enter the system" type="Sequence Episode">
  <SubEpisodes>
    <SubEpisode>Operator turns off the machine</SubEpisode>
    <SubEpisode>ATM system turns off</SubEpisode>
  </SubEpisodes>
</Episode>
<Episode name="ATM system turns off" type="Alternation Episode">
  <SubEpisodes>
    <SubEpisode>ATM machine is turned off successfully</SubEpisode>
    <SubEpisode>ATM machine couldn't be turned off</SubEpisode>
  </SubEpisodes>
</Episode>
<Episode name="ATM machine is turned off successfully" type="Sequence Episode">
    <SubEpisodes>
      <SubEpisode>Operator cuts the connection to the bank</SubEpisode>
      <SubEpisode>Operator removes deposited envelopes</SubEpisode>
      <SubEpisode>Operator replenishes cash and paper</SubEpisode>
    </SubEpisodes>
</Episode>
<Episode name="Operator inputs administrator password" type="Atomic Episode" />
<Episode name="Operator cuts the connection to the bank" type="Atomic Episode" />
<Episode name="Operator turns off the machine" type="Atomic Episode" />
```

```
<Episode name="Operator removes deposited envelopes" type="Atomic Episode" />
<Episode name="Operator replenishes cash and paper" type="Atomic Episode" />
<Episode name="ATM machine couldn't be turned off" type="Atomic Episode" />
<Episode name="Operator is not allowed to enter the system" type="Atomic Episode" />
</UseCase>
</EpisodeModel>
```

B.3 Event Level

```
<EventModel>
    <PrimitiveEvent name="Display Message" type="Response" />
    <PrimitiveEvent name="Input Code" type="Stimuli" />
    <PrimitiveEvent name="Timeout" type="Response" />
    <PrimitiveEvent name="Validate Code" type="Action" />
</EventModel>
```