Comprehension and Transformation of Object-oriented Models

Zeng Zi

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montreal, Quebec, Canada
March 2007

© Zeng Zi, 2007

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-28959-4
Our file Notre référence
ISBN: 978-0-494-28959-4

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

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

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

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

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

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

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

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

Canada

Abstract

Comprehension and Transformation of Object-oriented Models

Zeng Zi

During object-oriented (OO) software development, the problem domain is mapped into the solution space implemented by a programming language and executed by a computer system. During OO design, the real-world objects are mapped into software objects with assigned responsibilities to fulfill certain tasks. In order to improve the quality of software systems, a number of approaches have been proposed to improve the quality of the existing code. One of these is "restructuring", a process of improving the internal structure of a software system without altering its external behaviour. However, refining implements artifacts tends to be much more expensive than refining design artifacts. In addition, Model Driven Architecture (MDA) with its supporting tools has become the mainstream in software development and provides increasingly powerful facilities to automatically generate documentation and code from the platform-independent design model. These facts make the quality of a software product greatly associated with the quality of the design model. In recent years, there is a trend of addressing restructuring at a higher level of abstraction. In MDA, the Unified Modeling Language (UML) model is widely used to build and visualize the design in a platform-independent way. In this dissertation, we propose the development of approaches to obtain comprehension, and perform restructuring of the UML design model.

Acknowledgments

I would like to acknowledge many individuals who have provided help for this dissertation.

First, I would like to express my special gratitude to my supervisor, Dr. Constantinos Constantinides, for his help, guidance and support during my study at Concordia, and especially to this research work.

In addition, I thank Venera Arnaoudova, Hamoun Ghanbari, Laleh Mousavi-Eshkevari, Elaheh Safari, Paria Parsamanesh, for their valuable comments over my research work.

I would like also to express my thanks to Guillaume Theoret, for his cooperation while working on the supporting tools. I also want to thank Amir Abdollahi Foumani for his valuable advice throughout this project.

At last, but not least, I would like to thank my parents, who supported and encouraged me during my studies in Concordia. I also want to thank my boyfriend for his encouragement, support and tolerance during my research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The maturities of object-oriented software development technologies and process models such as the Unified Process have managed to bridge the gap between the real world and the software models, having also significantly reduced the risk of development. Compared to the wide availabilities of code generation tools, supporting tools for automatically improving an existing design model are rare. This dissertation aims to fill this gap and to provide strategies and supporting tools to aid in comprehension and improvement of a design model through model transformation, knowledge representation, refactoring and design patterns technology as a mean to enrich OO design. The expected benefit will be an implementation whose quality is the result of the natural mapping of the design, rather than an implementation which would need significantly corrective measures through refactoring.

1.1 Major contributions

The expected contributions of this research are as follows:

1. To provide a human readable and computer tractable representation of a UML model in both the static and the dynamic view for the purpose of automatic analysis and modification.

2. To implement comprehension strategies and analysis algorithms in forms of higher order queries which utilize the advantages of database logic processing.

3. To implement automatic restructuring of a model by simple text processing.

4. To provide automation that can support 1-3.

1.2 Synopsis of the dissertation

The rest of the dissertation is organized as follows: In chapter 2, we introduce the theoretical background of the thesis. In chapter 3, we discuss the problem and motivation behind this research. In chapter 4, we present our proposal of this research. In chapters 5, 6 and 7, we discuss our methodology. In chapter 8, we provide a case study to illustrate our approach and the deployment of our tool. In chapter 9, we discuss related work in comparison with our proposal and provides an evaluation of this dissertation. In chapter 10, we provide our conclusions and recommendations for future work.

Chapter 2

Theoretical background

In this chapter we discuss the necessary theoretical background to this research.

2.1 Model Driven Architecture

Model Driven Architecture (MDA) [Groa], is an initiative by the Object Management Group (OMG), a consortium of companies, in defining an approach to software development based on modeling and automated mapping of models into implementation. The fundamental MDA pattern includes the definition of a platform-independent model (PIM) and the corresponding automated mapping to one or more platform-specific models (PSMs) [CH]. In MDA, the terms PIM and PSMs refer to models of a software system which may or may not link to a specific technological platform, such as a specific programming language. For example, a generic description (e.g. a text description) of a software system can be considered

as a platform-independent model, while a representation of a software system using Java or C++ can be considered as a platform-specific model. The motivation is to model a system in a platform-independent way. Model Driven Architecture focuses on the functionality of the system rather than addressing their implementation. New object-oriented software development implementation technologies for business functions do not require repeating modeling, as modeling can be performed in a platform-independent way. As a result, system functionalities are modeled only once. For example, a system can be modeled and implemented in Java. If for some reasons it should be re-implemented in C++, the design model may have to be restructured if it is platform-specific. Therefore, if the system is modeled in a platform independent way, the model can be deployed regardless of the technology in which it is implemented.

2.2 The Unified Modeling Language

Born out of the unification of earlier object-oriented graphical model languages, the Unified Modeling Language (UML) [BRJ99] is a general purpose graphical modeling language capable to capture and visualize the real world system based on object definitions and object relationships. The UML is an open standard under the control of OMG. A system can be modeled through a combination of static and dynamic views, supported by a number of different artifacts. Two artifacts widely used are the class diagram, and the two types of interaction diagrams

(communication and sequence diagram). The class diagram can represent the static view of the system and the interaction diagrams can collectively represent the dynamic view.

2.3 Use-case driven development in object-oriented development

Use-case driven development is a term adopted to stress the notion of the use-case model as a central theme driving the OO development process. The use-case model is used to capture functional requirements in scenarios [Figure 1]. Based on the narrative description of use-case scenarios, a domain model captures real-world concepts and their attributes, as well as the associations between concepts. Translated from a given use-case scenario, a system sequence diagram (SSD) treats the system as a black-box and illustrates: 1) the request events that an external actor generates (implicitly illustrating the corresponding operations requested from the system), 2) the order of these events and 3) inter-system events (if any). In an SSD, an actor issues a sequence of request events, each invoking a corresponding operation at the system end. The set of system operations that correspond to an SSD constitute a subset of the interface (the overall behaviour) of the system. Each system operation can be associated with a set of formal or semi-formal specifications, referred to as a system operation contract, which is implemented as

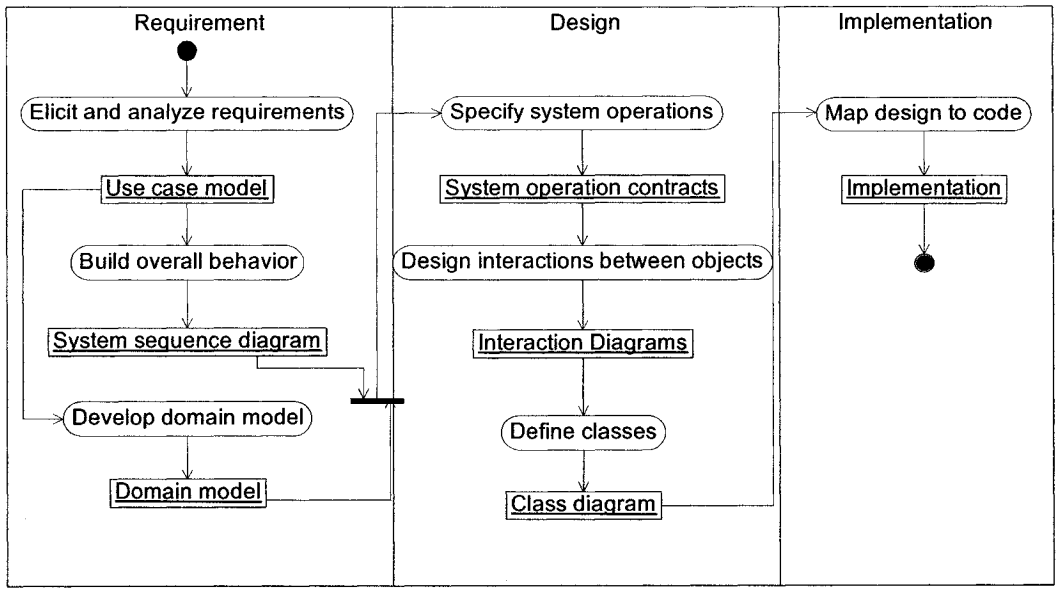

Figure 1: UML activity diagram illustrating use-case driven development.

an interaction diagram. During this activity, developers may apply certain guidelines on the assignment of responsibilities to objects through the deployment of responsibility patterns [Lar04], and design patterns (GoF) [GHJV95] in order to reuse proven designs and experience. In an interaction diagram, upon reception of the external message, the subsystem under design must provide a logical solution to the problem by illustrating how objects collaborate in order to fulfill this responsibility. To perform this collaboration, objects interact via message passing. Based on the domain model and interaction diagrams, a class diagram illustrates software classes (containing state and behaviour) and their associations. The class diagram and the set of interaction diagrams are mapped to the implementation model (potentially to any object-oriented language).

2.4 Production rules

The Unified Modeling Language provides a graphical model for the system under development. The semantics and metadata behind the model could be expressed by a set of abstract rules which we refer to as production rules (PR) [FCa] that are finite according to their definition. A production system can be utilized as a language to represent knowledge.

2.5 Comprehension

In the context of software development and maintenance, comprehension is the process of understanding the functionality as well as providing measures towards the quality of a software system. Comprehension is directed toward the entire set of artifacts that comprise the software system.

2.6 Model transformation

In MDA, the term "model transformation" refers to: 1) the transformation of a model based on a metamodel into another model which is based on another metamodel (e.g. model transformation between Eclipse Modeling Framework (EMF) models [EMF]) or 2) the transformation of a model to another model which is based on the same metamodel. An example of the second kind of model transformation is "restructuring". Generally speaking, restructuring refers to : a change

made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour [Fow99]. Traditional restructuring focuses on the implementation as the primary artifact. Currently, restructuring to find "bad smells" and restructuring to patterns [Ker04] are popular among researchers and practitioners and they are deployed as a guidance for code improvement.

In this chapter, we discussed several topics which serve as the theoretical background to this research.

Chapter 3

Problem and motivation

In this chapter we discuss the problem and the motivation behind this research which constitutes the scope of this dissertation.

With the support of a collection of tools, MDA-based software development proceeds through the stage of modeling with UML (platform-independent model) to automatic documentation generation or code generation (from platform-specific models). During these stages, design is the activity in which a real world domain model is mapped into a software model where objects are assigned responsibilities to fulfill requirements. The code, most of which is currently skeletal, can be automatically generated from the design model through mapping. The implementation artifacts are completed by implementing each functionality modeled in design stage using desired technologies (e.g. Java, C++). Therefore, in MDA, design produces critical artifacts in the development process and it greatly affects

the quality of software products. However, anomalies in design such as high coupling, low cohesion and bad information hiding are bound to affect the quality of the end-product. Moreover, they tend to be difficult to detect and address. In addition, experience has shown that fixing errors in design is less expensive than fixing errors in code [BPM]. Since traditional improvement toward the software system is to improve the code, approaches and supporting tools are strongly desired by the developers for detecting and addressing anomalies early in the design stage. However, the UML representation of the static and dynamic model is not comprised of a single artifact. The representation is multi-dimensional and it is comprised of a collection of artifacts. For medium- to large-scale systems, the UML representation will contain a large collection of artifacts, which will make their comprehension and possible transformation difficult, tedious and error prone. Thus, to perform these activities automatically, the design model should be described in a machine-manipulable form that can be automatically analyzed and restructured (if needed). In addition, in which way the model can be analyzed is an open issue. Furthermore, there is also the issue of how to make necessary modifications and how these modifications can be mapped into a new model. The motivation behind this research is described by the following points:

1. To represent the entire model (static and dynamic view) in a single form which is appropriate for easy manipulation in order to extract knowledge and achieve comprehension.

2. To develop strategies (algorithms) for knowledge extraction in order to identify anomalies.

3. To develop strategies (algorithms) to restructure the model along the lines of "bad smells" and "refactoring" [Fow99] strategies and along the lines of "restructuring to patterns" [Ker04].

4. To support automation for 1-3.

In this chapter, we discussed the scope of this dissertation in terms of problem and motivation.

Chapter 4

Proposal

In this chapter, we discuss our research proposal. The main idea is illustrated in Figures 2 and 3.

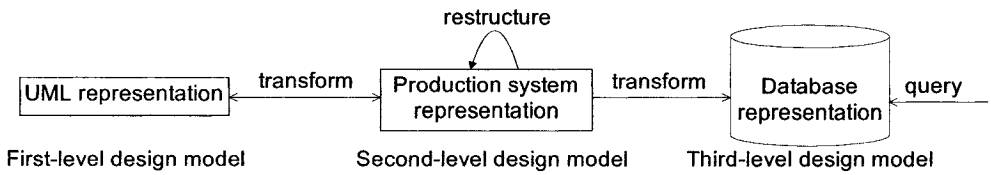

Figure 2: Three level design models.

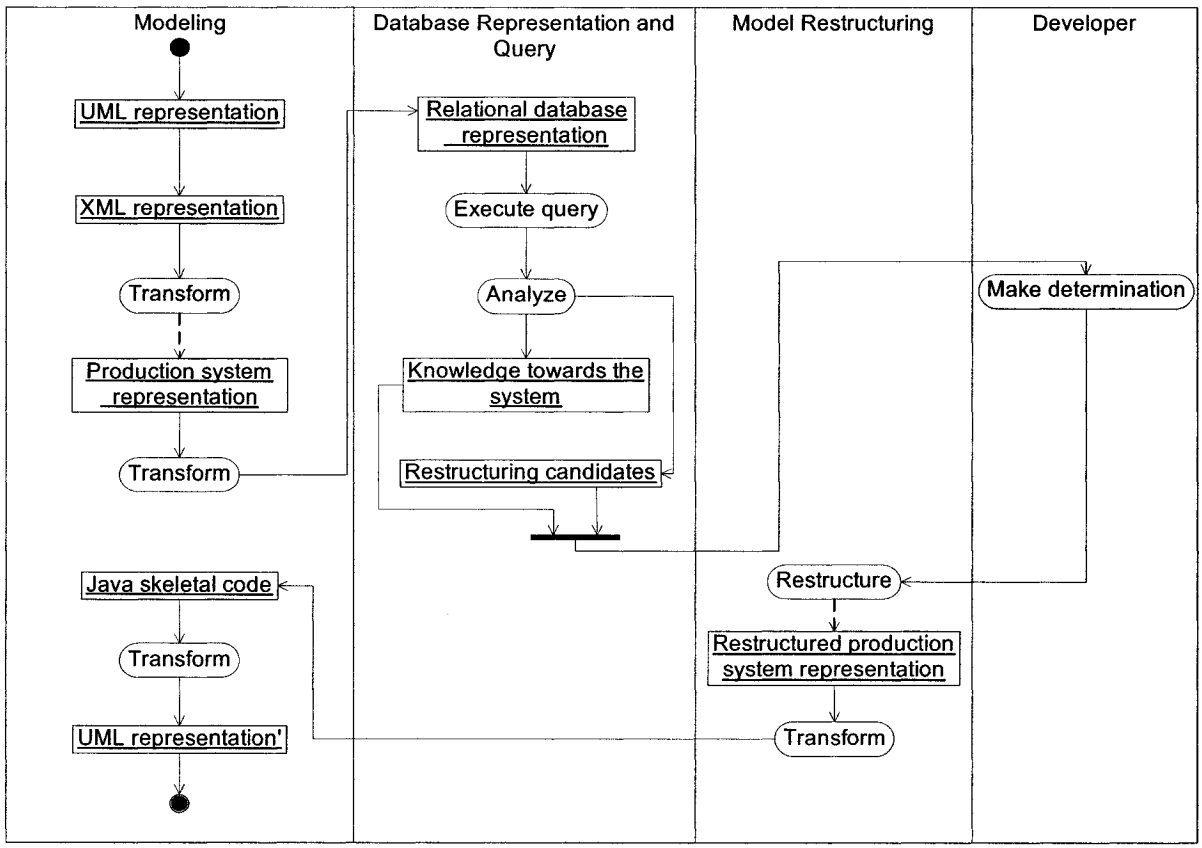

Figure 3: UML activity diagram illustrating the steps of the proposal.

4.1 Model transformation into a production system

To obtain an alternative representation of a UML model, we deploy a model transformation technique. The motivation behind this idea is to transform a model to another one which is easier to manage, and is able to be automatically analyzed and modified. XML Metadata Interchange (XMI) [Grob] is used as an interchange format for a UML model. Also, it is commonly used as an intermediate model representation through which code and documentation can be generated from the UML model. XMI can also be considered as a textual representation of the model. One approach to automatically manipulate a UML model is to manipulate its corresponding XMI representation. However, as a data exchange facility, the XMI representation of a model often contains superfluous information such as tags and graphic positions. Therefore, even if there are plenty of tools available that could generate XMI from a UML model, we choose to target the transformation of the model into an alternative representation which can be easily read by humans and easily manipulated by a machine. Our chosen approach is the notion of production rules, which describes an abstract concept language for representing the semantics and metadata behind the model.

We refer to the UML model as the first-level design model, and the production system representation generated from UML as the second-level design model. In

addition, the second-level design model serves as an intermediate model representation, which is expressed at a higher enough level to be humanly readable and at a lower enough level to be machine manipulatable. To obtain comprehension of a system, we define syntax and semantics to represent the two dimensions of the model: static and dynamic. The UML models, involved in the model transformation process are the UML class diagram and interaction diagrams, which address the structure and the behaviour of the system respectively.

4.1.1 Representation and storing the second-level design model in a relational schema

The second-level design model provides a complete and manageable description (static and dynamic) of the system. Analysis over this model can be carried out to identify anomalies which refer to any factor that affects the quality of the system, also refer to "bad smells" [Fow99] in context of code. However, when encountering the situations where data of a particular category needs to be extracted and analyzed, we need to decode the production system representation, classify the data and then extract requested information. This text processing can become tedious for a large system. Data describing the design model can be clearly classified and transformed into the relational database. For instance, a model would have classes, a class would have features (attributes and methods), and each feature would have properties, all of which can be modeled into a relational schema by

a collection of tables. Moreover, analysis toward the model is usually based on logical reasoning which could be implemented in a higher order query language. In this approach, we can combine the storage mechanism with the logical system: a combination supported by the database technique. Therefore, in general we can translate the production system representation into a database, in which the information and data of the model are classified and organized into an appropriate relational schema. We refer to the database representation as the third-level design model. Analysis can be performed by applying proper strategies (algorithms) through executing statements in the form of queries (possibly with the assistance of additional programming if required).

4.1.2 Extracting knowledge and achieving comprehension

Comprehension of the model could be manually performed by drawing observation over the UML diagrams. However, in situation where 1) there are many classes with many features, 2) the interaction among objects is complex and a potentially large collection of interaction diagrams is required to be analyzed, a manual comprehension can be tedious and error-prone over these artifacts. Even if proper algorithms have been applied to help the comprehension of a system, the problem is that the input data for the algorithm might not be correct due to the loss, or misunderstanding of information towards the model by manual analysis.

Our production system produces a single representation of the entire model originated from the UML class diagram and a collection of UML interaction diagrams. Therefore, we propose several strategies through which the production system representations can be analyzed by extracting knowledge from the third-level design model to achieve comprehension.

4.1.3 Restructuring

There are three possible places where we can modify a design: 1) Modify UML diagrams manually (first-level design model), 2) Modify the production system representation (second-level design model) and transform it back to UML, 3) Modify database (third-level design model) and transform it back to UML. All of these approaches are feasible but not all of them are straightforward. The first approach requires that developers read and understand the analysis results from database queries, then manually modify the UML diagrams. This can be tedious and error prone. It also implies that the process cannot be automated. Thus, both (2) and (3) seem more viable solutions. We plan to perform modification over the second-level design model. Furthermore, there may be the case that without analyzing the model, developers decide to re-design part of the model. In this case, modification of the model could be carried out directly on the second-level design model without relying on the results of analysis from the database. The restructured second-level

design model can be translated back to corresponding UML model. Our objective is to define a system that can guide developers to perform the correct and necessary restructuring towards the second-level design model, which will lead to an improvement of the design. Ideas for this approach are borrowed from "software refactoring" [Fow99]. We feel that some categories of refactoring (including refactoring to patterns), which are discussed in the literature in the context of implementation can be applicable in our proposal and be deployed in the context of design representation.

4.2 Tool support

In this section, we discuss applicable tools we deploy as well as the desired tools we plan to build as Figure 4 shows.

4.2.1 Tools to deploy

Currently, there are a number of different tools to support modeling frameworks and code generation facilities, for example EMF, Umbrello UML [umb] and PoseidonUML [Pos]. The idea behind the model transformation from UML to code is to utilize XML. As an OMG standard for exchanging metadata information via XML, XMI can be generated by many modeling tools such as EMF, and can be used to generate code corresponding to a UML model. Similarly, XMI could

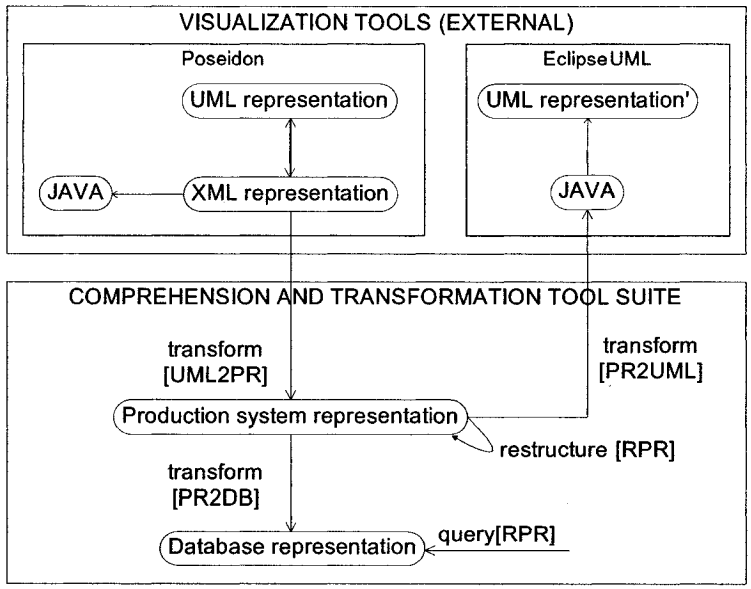

Figure 4: Automation and tool support.

be used to automate the process of bi-directional transformation between UML and production system representation. That is to say, we can take UML as an input, parse the XMI representation generated by an existing tool, and take a production system representation as an output. However, there is certain shortcoming existing in EMF. The generated XMI representation uses a UML2 library which is not openly available. This problem makes EMF difficult to work with. PoseidonUML is one of the tools that seem to be a viable technology without the problems encountered with EMF. We plan to deploy Poseidon to aid in the transformation from the UML representation to the production system representation. The transformation can be achieved by passing through the entire XML and generating the production system representation of the model. To transform the

production system representation back to the UML representation, XMI is not a good choice since it is extremely complex to generate. The reasons are as follows: 1) XML file is verbose and tedious. Even a small model can translate into a large XMI representation. Therefore, generating the production system representation of a model back to its corresponding XMI representation is obviously difficult. 2) A valid XMI representation that could be used to produce UML diagram would have to include additional information such as graphic positions, which can not be obtained from the production system representation. The reason for this is that for the versions of XMI representation which do not use UML2 library, because during the parsing of XMI representation of a model, we get rid of information that is not related to the model itself such as graphic position, but is required if we want to import XMI back to UML. As an alternative approach, we plan to generate Java skeletal code from the production system representation which can be imported into visualization tools in order to generate a refined UML model. We can deploy EclipseUML [Ecl] in this approach, which is a plug-in for Eclipse that allows building UML models, and can support reverse engineering of Java code.

4.2.2 Tools to develop

To facilitate the model representation, comprehension and transformation, we plan to develop the following tools listed below according to the respective activity which they perform. These tools constitute our comprehension and transformation tool

suite (Figure 4).

UML to production system representation (UML2PR) Taking a UML model as an input, XMI representation could be generated automatically by Poseidon UML. To generate a production system representation, we plan to build a tool to parse the XMI representation and transform it to the production system representation via text processing. This tool will be developed in Ruby, which is very good at text processing since it supports regular expressions.

Production system representation to database schema (PR2DB) To store the data of the production system representation of the UML model, we plan to build a parser to decode the production system representation, classify the data, and store it into a relational database schema via text processing and JDBC processing. This tool will be developed in Java.

Restructuring of production system representation (RPR) To automate the process of restructuring the second-level design model, we plan to develop a tool to obtain the restructuring candidates and to restructure the production system representation based on the proposal, via text processing and JDBC processing. This tool will be developed in Java.

Production system representation to UML (PR2UML) To translate the restructured second-level design model back into a new UML model, we propose to develop a tool that generates Java skeletal code via text processing,

which can be imported into EclipseUML, and generate a refined UML class diagram and sequence diagrams. This tool will be implemented in Ruby.

In this chapter we discussed the proposed methodology by introducing the structure of the three-level design model, the approach to comprehension and the model transformation, and the supporting tools to deploy and to develop for the provision of automation.

Chapter 5

Production rules: Defining the second-level design model

In this chapter, we discuss how production rules can be deployed to provide a representation of the model.

In the following subsections, we define the syntax and semantics of the production rules to represent the static and dynamic views of the system respectively by extending the original definition of production rules presented in [FCa]. In [FCb], the authors define production rules as a set of abstract rules which supports the semantics of object-oriented artifacts, G. G is defined in terms of a set of five elements, each of which is finite. Let G = C, A, M, P, R, such that:

1. C is a set of classes.

2. A is a set of attributes.

3. M is a set of methods.

4. P is a set of transformation rules that defines an object oriented design semantics in terms of: a) definition of classes, b) hierarchy of classes, c) relationships between classes, d) system scenarios in terms of message passing between classes.

5. R is set of relationships and concepts defined by the object-oriented methodology. We define this set as `[declare]`, `[has]`, `[call]`, `[extend]`, `[declare/receive]`, `[set]`, `[supplement]`.

From the original definition of the production system, we can see that it can not describe all model informations in both the static and the dynamic views. For example, there is no support for some essential properties of the model such as `visibility`, `return type`, `static`, `abstract`, `associations` etc. To faithfully represent the model, the original production rules should be extended. We refine and extend production rules to be a set of abstract rules that supports G. G, which describes the semantics of the UML model artifacts (class diagram and interaction diagram), is represented by a set of elements, each of which is finite.

Let G = {C, Attr, M, Assocs, I, P, R}, such that

1. C is a set of classes.

2. Attr is a set of attributes.

3. M is a set of methods.

4. `Assocs` is a set of associations.

5. `I` is a set of interactions.

6. `P` is a set of transformation rules that defines an object-oriented design semantics in terms of: a) definition of classes, b) definition of attributes, c) definition of class-methods (without method body), d) associations between classes e) scenarios in terms of message passing between classes.

7. `R` is set of relationships and concepts defined by the object-oriented methodology. We define this set as `[Class]`, `[Property]`, `[Operation]`, etc.

The reasons for adopting production rules are: 1) The representation is much more terse than the XMI representation since it contains only necessary information required to describe the model. To describe a model, we must to extract knowledge from the system. A set of production rules that support `G`, can be defined to faithfully represent the UML static and dynamic model, for the purpose of system comprehension. 2) Usage of production rules can be "infinite." The syntax of the production rules is defined to be "finite", but they can describe infinite kinds of models through building of derivation sentences over corresponding production rules.

We define production rules to describe the static structure of the system represented by a class diagram in Tables 1 and 2. In Table 3, we define production

rules to support dynamic behaviour of the system represented by communication/sequence diagrams.

There are six parts in a production system representation according to G (discussed in proposal): 1) List of classes 2) List of enumeration 3) Values in enumerations 4) Definition of classes and their properties 5) Definition of associations 6) Scenarios described by showing interactions among objects. Parts 1-5 could be defined by providing the class diagram as an input. Part 6 calls for more careful consideration. The descriptions of the use-case scenarios are organized one by one, based on the use-case diagram. In a use-case scenario, system operations are ordered based on the SSD. The interaction description, in production system representation, describes each use-case scenario with all system operations involved which is ordered together with sequences of their internal operations. In addition, each system operation will be illustrated by `Begin Interaction<InteractionName>` and `EndInteraction<InteractionName>`. The default `InteractionName` is the name of the corresponding interaction diagram. The name of each interaction diagram should be the name of corresponding system operation.

In this chapter we discussed the definition of the production system deployed in this research.

PRODUCTION	DESCRIPTION
Boolean	Datatype: Boolean.
Real	Datatype: Real.
Integer	Datatype: Integer.
String	Datatype: A sequence of characters.
Set	Datatype: A collection of non-redundant elements.
OrderedSet	Datatype: An ordered set.
Bag	Datatype: A collection of unordered elements, allowing redundancies
Sequence	Datatype: A collection of ordered elements.
T	Datatype previously defined.
C::=[Class]	C is a class.
<ClassList>::=<C+>	A non-empty list of classes.
<C>::= [Visibility] <Public|Protected|Private> [IsAbstract] [IsFinal] [IsInterface] [extends]<C'> [Implements]<C'>	Class definition. If not abstract, eliminate this keyword. If not final, eliminate this keyword. If not interface, eliminate this keyword.
<C>::=[Property]<Attribute>	C has Attribute.
C.Attribute	Attribute is a field of C.
<C.Attribute> ::= [Type] <T> [IsStatic] [IsFinal] [Visibilty] <Public|Protected|Private>	Attribute definition. Attribute is of type T. If not static, eliminate this keyword. If not static, eliminate this keyword.
<C>::=[Operation]<operation()>	C has operation().
C.operation()	operation() is a feature of C.

Table 1: Productions for static definitions (Part1).

PRODUCTION	DESCRIPTION
<C.operation()> ::= [Visibility] <Public|Protected| Private>	Operation definition.
[IsStatic]	If not static, eliminate this keyword.
[IsAbstract]	If not abstract, eliminate is keyword.
[ReturnType]<T>	If void, make it blank.
[Parameter]<String*>	List of Parameters, seperated by commas.
[IsFinal]	If is not, eliminate this keyword.
<AssociationList>::=<associationName+>	A non-empty list of associations.
<associationName>::=String	associationName is literal.
<Set | OrderedSet | Bag | Sequence> :: =[Type]<T>	A collection of elements of type T can be a set, ordered set, bag, or sequence.
Enumeration	An enumeration type.
<AssociationName> ::= [Reference]<C1> [Lower-bound]<n1> [Upper-bound]<n2> [Reference]<C2> [Lower-bound]<n1> [Upper-bound]<n2> [Relationship| <Composite| Aggregate | unidirection | bidirection> [Type]<Set | OrderedSet | Bag | Sequence>	Multiplicity in AssociationName over C1 end is between n1 and n2, where n is an integer. As above for C2.
Composition	Predefined association, composite.
Aggregation	predefined association, aggregate.
unidirection	Predefined association, unidirection.
bidirection	Predefined association, bidirection.

Table 2: Productions for static definitions (Part2).

PRODUCTION	DESCRIPTION
<C.operation()>[Parameter]<T1,T2..>::=	C.operation() with Parameter T1,T2
[calls]	calls
<B.operation()>[Parameter] <T1,T2..>[iter]<n>	B.operation() with parameters of type T1, T2... n times.

Table 3: Productions for dynamic definitions.

Chapter 6

Defining the third-level design model

In this chapter we discuss the definition of the third-level design model captured by the relational database schema mapped over from the second-level design model.

The production system representation is organized in a way that it describes the structure of the model. The ecore [BSM+] model serves as a metamodel for any EMF model and uses `EClass, EAttribute, EReference Classes, EOperation` to represent all classes, attributes, references and operations in a model. Borrowing ideas from ecore model, we define the relational database schema which contains the following parts essential for the comprehension of the system and includes both static and dynamic views of the model: `Class, Attribute, Operation Association and Interaction`. Each element is represented by a table since it

has its own properties. `Class, Attribute, Operation, and Association` together represent the static model of a system. Table `Interaction` represents the dynamic view of a system. The database schema is shown in Tables 4 - 8. The entity relationship (ER) diagram in Figure 5 describes the structure and entities (with their properties) of the schema. Although most of the data could be extracted from the second-level design model directly and inserted into the relational database schema, some of them are not so straightforward. There are a few points that need to be elaborated on concerning the `Interaction` table:

1. The `SeqNumber` in the table indicates the sequence number that implies to the order of the interaction. For example, we have a set of productions representation which describe a system operation as follows:

```
Begin Interaction<Op1>
<A.Op1()>[Parameter]<>::=[calls]<B.Op2()>[Parameter]<Integer>
                      ::=[calls]<B.Op3()>[Parameter]<>
<B.Op2()>[Parameter]<Integer>::=[calls]<C.Op4()>[Parameter]<>
End Interaction<Op1>
```

   The `SeqNumber` for `A.Op1(), B.Op2(Integer), C.Op4()` is 1, 1.1, 1.1.1, 1.2 respectively. We can see that the way we construct the `SeqNumber` is the same as the way we construct the sequence number in the communication diagram which is one type of the UML interaction diagram [Lar04].

```
Begin Interaction<Op1>
<A.Op1()>[Parameter]<>::=[calls]<B.Op2()>[Parameter]<Integer>[iter]<5>
...
End Interaction<Op1>
```

Class				
field	**type**	**modifier**	**key**	**definition**
ClassName	VARCHAR	NOT NULL	primary	
Visibility	VARCHAR	NOT NULL		public/protected/private
IsAbstract	VARCHAR	NULL		"true" if abstract, else "false"
Implements	VARCHAR	NULL		Else put NULL
Extends	VARCHAR	NULL		Else put NULL
IsInterface	VARCHAR	NULL		Else put NULL
IsFinal	VARCHAR	NULL		"true" if final, else "false"

Table 4: Database schema for Class.

Attribute				
field	**type**	**modifier**	**key**	**definition**
AttributeName	VARCHAR	NOT NULL	primary	
Type	VARCHAR	NOT NULL		
ClassName	VARCHAR	NOT NULL	foreign	
			primary	
IsStatic	VARCHAR	NULL		"true" if static, else "false"
IsFinal	VARCHAR	NULL		"true" if final, else "false"
Visibility	VARCHAR	NOT NULL		public/protected/private

Table 5: Database schema for Attribute.

2. Elements under `OperationNames` are ordered following the calling sequence of internal operations for each system operations.

In this chapter, we discuss the relational database schema to support the storage of the model.

Operation				
field	**type**	**modifier**	**key**	**definition**
OperationName	VARCHAR	NOT NULL	primary	
ReturnType	VARCHAR	NOT NULL		If not void, put its type, else put NULL
ClassName	VARCHAR	NOT NULL	foreign	
			primary	
Visibility	VARCHAR	NOT NULL		
IsAbstract	VARCHAR	NULL		"true" if abstract, else "false"
IsStatic	VARCHAR	NULL		"true" if static, else "false"
IsFinal	VARCHAR	NULL		"true" if final, else "false"
Parameter	VARCHAR	NULL	primary	A list of parameters

Table 6: Database schema for Operation.

Interaction				
field	**type**	**modifier**	**key**	**definition**
OperationName	VARCHAR	NOT NULL	foreign	
ClassName	VARCHAR	NOT NULL	foreign	
SeqNumber	VARCHAR	NOT NULL	primary	
InteractionName	VARCHAR	NOT NULL		
Looping	Integer	NOT NULL		The iterations of an operation
Parameter	VARCHAR	NULL	foreign	A list of parameters

Table 7: Database schema for Interaction.

Association				
field	**type**	**modifier**	**key**	**definition**
AssociationName	VARCHAR	NOT NULL	primary	
end1	VARCHAR	NOT NULL	foreign	
			primary	
Lowerbound_end1	VARCHAR	NOT NULL		
Upperbound_end1	VARCHAR	NOT NULL		
end2	VARCHAR	NOT NULL	foreign	
			primary	
Lowerbound_end2	VARCHAR	NOT NULL		
Upperbound_end2	VARCHAR	NOT NULL		
Relationship	VARCHAR	NOT NULL		Composite/Aggregate /unidirection/bidirection
Type	VARCHAR	NULL		Set/OrderedSet /Bag/Sequence

Table 8: Database schema for Association.

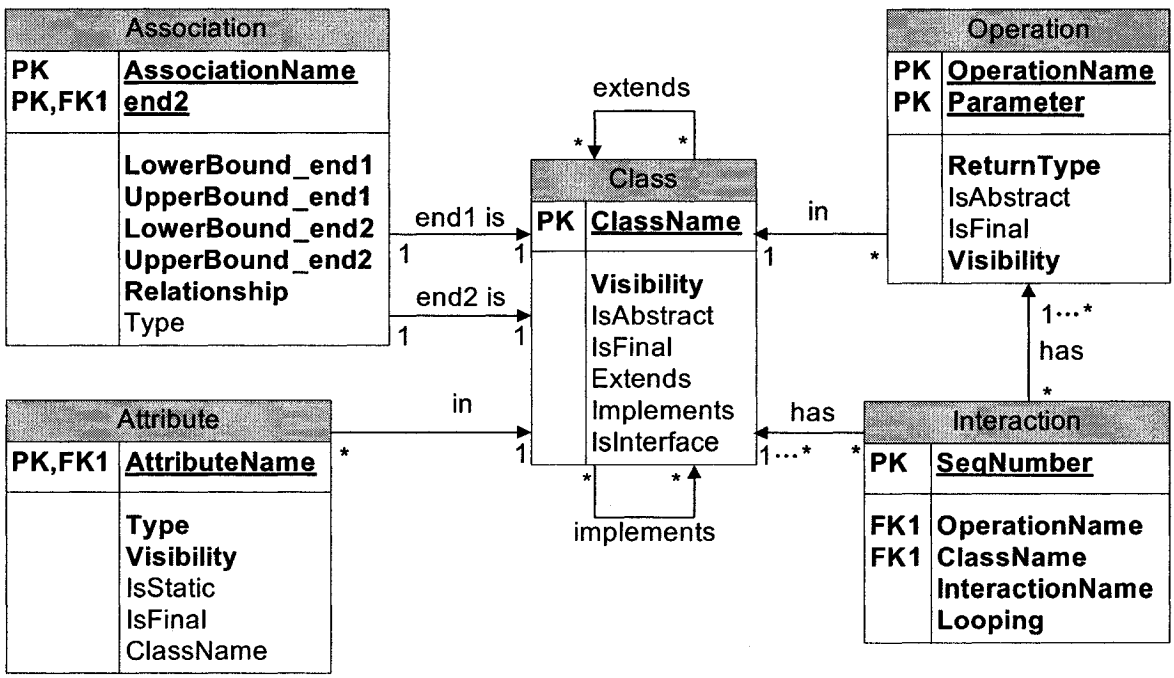

Figure 5: ER diagram for the relational database schema.

Chapter 7

Comprehension and transformation

In this chapter we discuss the extraction of knowledge from the relational database schema, in order to achieve comprehension of the model. We also discuss strategies in order to restructure the production representation of a model.

7.1 Extracting knowledge to obtain comprehension

Basic knowledge of a design model is easy to obtain by observing the UML diagram manually. However, more complex knowledge from the model that is not straightforward to extract or requires analysis over a large group of entities, adds

difficulties to perform the comprehension process manually. We will illustrate a number of cases which tend to be tedious to achieve comprehension by manually examining the UML diagram but could be easily performed by executing statement in forms of query over the third-level design model: 1) To identify "lazy class", 2) To identify "data class", 3) To identify unused operations, 4) To trace the calling sequence of messages in a given use-case scenario, 5) To check for consistency between the static and the dynamic model, 6) To measure coupling and cohesion of the model.

7.1.1 Identifying a "lazy class"

In [Fow99], a "lazy class" is defined as one that does not do enough. At the design level, responsibilities of a class are indicated by their features. Identifying a potential lazy class could be achieved by examining the number of their features. However, the number of the features of a class would not necessarily indicate that it is a lazy class. For example, a class may have only one method which takes a possible large number of responsibilities. Since the responsibilities taken by the method cannot be definitely described by the information provided in the UML model, we can only indicate potential lazy classes in the model. An empty class, which is not categorized to be a lazy one, needs to be identified separately. In addition, a class defined in the static model but never used in the dynamic model is a strong candidate to be a lazy class. Following this idea, we thus indicate a

lazy class if 1) a class has only one method, 2) a class is empty, or 3) a class is unused. We distinguish between a lazy class in an inheritance relationship (in which we suggest to collapse the hierarchy) and one which is not in an inheritance relationship (in which we suggest to delete and transfer its responsibility to another class). The potential lazy classes could be found by the following queries:

Case1: A class has only one method:

1. First, create a view `AllLazy` to find out all classes that have only one method.

```
SELECT   o.ClassName as LazyClass, count(*) as numOfMethod
FROM     Operation o
GROUP BY LazyClass
HAVING   count(*) = 1;
```

2. Identifying the classes in an inheritance relationship from above view `AllLazy`.

```
SELECT l.LazyClass
FROM   AllLazy l, class o
WHERE  l.LazyClass = o.ClassName AND o.extends!='';
```

Case2: A class is empty:

```
SELECT ClassName AS EmptyClass
FROM   Class
WHERE  ClassName
NOT IN (SELECT ClassName
        FROM   Attribute)
AND     ClassName
NOT IN (SELECT ClassName
        FROM   Operation);
```

Case3: A class is never used:

The following query creates a view `AllUnUsed` to identify classes which are defined in the static model but are not used in the dynamic model.

```
SELECT ClassName AS unusedClass
FROM   Class
WHERE  ClassName
NOT IN (SELECT ClassName
        FROM   interaction);
```

There are two cases to be considered when examining the result from the above query:

1. Identifying super classes in view `AllUnUsed`. A super class that is not referenced in the interactions is perhaps because of polymorphisms in an inheritance relationship. Therefore, more detail should be looked into to determine whether or not it is an unused class.

```
SELECT p.unusedClass AS unusedClassSuper
FROM   AllUnused p, class c
WHERE p.unusedClass=c.ClassName
AND    p.unusedClass IN (SELECT extends
                         FROM class);
```

2. Identifying interfaces. An interface is the most common case that a class is in the static view but not in an dynamic view.

```
SELECT p.unusedClass AS unusedClassInterface
FROM   AllUnused p, class c
WHERE  p.unusedClass=c.ClassName
AND    p.IsInterface='true';
```

7.1.2 Identifying a "data class"

In [Fow99], a "data class" is defined as one in which all of its features are attributes. This type of class would need further consideration to undertake some responsibilities. The following query identifies the data classes of the model.

```
SELECT ClassName AS DataClass
FROM   Class
WHERE  ClassName
NOT IN (SELECT ClassName
        FROM   Operation)
AND    ClassName
IN     (SELECT ClassName
        FROM   Attribute);
```

7.1.3 Identifying unused methods

An unused method is one defined in a class but never used in the dynamic model. Normally, this kind of method adds no value to the system but it is a potential place where errors might occur. We can identify unused methods by the following queries: (We exclude interface classes in this case since operations in an interface class will never appear in the dynamic model)

```
SELECT OperationName AS unusedMethod, o.ClassName, Parameter
FROM   Operation o, class c
WHERE  c.ClassName=o.ClassName
AND    c.IsInterface='false'
AND NOT EXISTS (SELECT o.OperationName, o.ClassName, o.Parameter
                FROM   interaction i
                WHERE  o.OperationName=i.OperationName
                AND    o.Parameter=i.Parameter
                AND    o.ClassName=i.ClassName);
```

7.1.4 Finding all aggregates of a given class

To further understand the relationship of classes, sometimes we need to find all aggregates of a given class, which can be obtained by the following query:

```
SELECT end1, end2
FROM   Association
WHERE  end1="GivenClass"
AND    Relationship="Aggregate";
```

7.1.5 Tracing the calling sequence of messages in a scenario

The calling sequence of messages in a scenario indicates how objects interact via ordered message passing. A scenario is described by a sequence of system operations that are described by sequences of internal operations, and the `InteractionName` in the `Interaction` table is the same as the system operation name. The parameters for this query are the name of each system operation with a particular order according to the SSD, for example `beginOp`, `doOp` and `endOp`. With the following query, we can get the calling sequence of messages of the scenario by finding out the calling sequence of their system operations with corresponding internal operations.

```
SELECT OperationName, ClassName, Parameter
FROM   interaction
WHERE  InteractionName='beginOp'
UNION
SELECT OperationName, ClassName, Parameter
FROM   interaction
WHERE  InteractionName='doOp'
UNION
SELECT OperationName, ClassName, Parameter
FROM   interaction
WHERE  InteractionName='endOp'
```

Once we get the calling sequence of messages of a scenario, we can use it to simulate the scenario, which can be later compared with the calling sequence traced from the code to check for consistency between model and implementation.

7.1.6 Checking for consistency between the static and the dynamic model

The dynamic model describes how objects, defined in the static model interact with each other via message passing. However, there are situations where an object would send a message to another object, against the definition in the static model. For example, it would be inconsistent that a method defined to be private called by objects, other than instance of the class in which it is defined. This strategies can be achieved by the following two steps:

1. Create a view `AllP` which obtains private operations list together with their `OperationName, ClassName, Caller` and `SeqNumber` by the following query.

```
SELECT i.OperationName, i.Parameter,i.ClassName,i2.ClassName
AS     Caller,i.SeqNumber
FROM   interaction i, operation o, interaction i2
WHERE  o.visibility = 'private'
AND    o.OperationName=i.OperationName
AND    o.Parameter=i.Parameter
AND    o.ClassName=i.ClassName
AND    instr(i.SeqNumber,'.')!=0
AND    i2.SeqNumber=reverse(right(reverse(i.SeqNumber),
       length(i.SeqNumber)-instr(reverse(i.SeqNumber),'.')));
 //the last two line are used to identify the Caller of a class
```

2. Identifying the methods that are used outside of the class in which they are defined:

```
SELECT DISTINCT OperationName, Parameter, ClassName, Caller
FROM   AllP
WHERE  ClassName<>Caller;
```

7.1.7 Measuring coupling and cohesion

In [PFC], the author argued that coupling and cohesion could be estimated in terms of "fanin" and "fanout." The terms define the number of messages that a given object receives and sends respectively, over a single execution path. An execution path is defined as an ordered sequence of message passing through which the interactions between objects could be captured. The authors also provide metrics to measure coupling and cohesion by tracing the source code. However, in the design model, some of the algorithms could also be applied to this proposal in order to measure these two factors. For example, class independency factor (CIF), method independency factor (MIF), class cohesion (CCH) factor and class coupling (CCP) factor. Each of the metrics is defined as follows:

$$CIF = \frac{class\ fanin}{class\ fanout}, \quad MIF = \frac{method\ fanin}{method\ fanout}$$

$$CCH = \frac{(number\ of\ classes\ in\ execution\ path)}{(number\ of\ methods\ in\ execution\ path)}$$

$$CCP = \frac{\sum Ci.fanin(Mi, Cj)}{\sum Cj.fanin(Mj, Ci)}$$

By analyzing the algorithm, we found that the parameters for all of them are: class fanin, class fanout, method fanin, method fanout, number of classes in execution path, number of methods in execution path, number of invocations for a method (e.g. `Ci.Mi()`) originated from a class (e.g. `Cj`) and number of messages received in a class (e.g. `Ci`) from another class (e.g. `Cj`). Our three level model transformation system can support those algorithms by providing the parameters listed above. The way through which we get these informations is discussed below. Since all these algorithms address problems in the scope of one use case scenario, the precondition is to find all related information with this scenario from the dynamic model. The user should inform which interaction diagrams are involved in this use case. Consider the following example: there are three interactions named `beginOp, doOp, endOp` involved in the current use case scenario, we can extract all related information corresponding to this scenario by the following query.

```
//create a view named "thisScen"
SELECT OperationName, ClassName, Parameter
FROM   interaction
WHERE  InteractionName='beginOp'
UNION
SELECT OperationName, ClassName, Parameter
FROM   interaction
WHERE  InteractionName='doOp'
UNION
SELECT OperationName, ClassName, Parameter
FROM   interaction
WHERE  InteractionName='endOp'
```

Class fanin and class fanout

1. class fanin : This factor could be obtained by executing the following query

```
SELECT ClassName, count(*) AS fanin
FROM
(SELECT ClassName,Caller, CallerOpPara, CallerOp,
        COUNT(*) AS Freq
 FROM
 (SELECT   i.OperationName, i.Parameter,i.ClassName,
           i2.ClassName AS Caller,
           i2.OperationName AS CallerOp,
           i2.Parameter AS CallerOpPara, COUNT(*) AS Freq
  FROM     thisScen i, thisScen i2
  WHERE  i2.SeqNumber=reverse(right(reverse(i.SeqNumber),
         length(i.SeqNumber)-instr(reverse(i.SeqNumber),'.')))
  AND      instr(i.SeqNumber,'.')!=0
  AND      i.ClassName!=i2.ClassName
  GROUP BY i.OperationName, i.Parameter,i.ClassName,
           Caller, i2.OperationName, i2.Parameter
  HAVING COUNT(*)>0) Cfreq
  GROUP BY CallerOpPara, CallerOp, ClassName, Caller
 HAVING COUNT(*)>0) Creq1
GROUP BY ClassName;
```

2. class fanout : We calculate class fanout by the following query:

```
SELECT ClassName, COUNT(*) AS fanout
FROM  (SELECT DISTINCT i.ClassName, ti.ClassName AS Callee,
       ti.OperationName AS CalleeOp,ti.Parameter AS CalleeOpPara
       FROM thisScen i, thisScen ti
       WHERE i.SeqNumber=reverse(right(reverse(ti.SeqNumber),
             length(ti.SeqNumber)-instr(reverse(ti.SeqNumber),'.')))
       AND   instr(ti.SeqNumber,'.')!=0
       AND   i.ClassName!=ti.ClassName) ft
GROUP BY ClassName;
```

Method fanin and method fanout

1. method fanin: The method fanin can be get in a similar way as we do with class fanin.

```
SELECT OperationName, ClassName, Parameter, COUNT(*) AS fanin
FROM
(SELECT OperationName,Parameter,ClassName,Caller,
 COUNT(*) AS Freq
 FROM
  (SELECT i.OperationName, i.Parameter,i.ClassName,
          i2.ClassName AS Caller,
          i2.OperationName AS CallerOp,
          i2.Parameter AS CallerOpPara,
          COUNT(*) AS Freq
   FROM thisScen i, thisScen i2
   WHERE i2.SeqNumber=reverse(right(reverse(i.SeqNumber),
         length(i.SeqNumber)-instr(reverse(i.SeqNumber),'.')))
   AND   instr(i.SeqNumber,'.')!=0
   AND   i.ClassName!=i2.ClassName
   GROUP BY i.OperationName, i.Parameter,i.ClassName,
            Caller, i2.OperationName, i2.Parameter
   HAVING COUNT(*)>0) Cfreq
 GROUP BY OperationName, Parameter,ClassName, Caller
 HAVING COUNT(*)>0) Cfreq1
GROUP BY ClassName,OperationName,Parameter;
```

2. method fanout: This is very similar with class fanout and it could be achieved as follows:

```
SELECT ClassName, OperationName, Parameter, count(*) AS fanout
FROM (SELECT DISTINCT i.ClassName, i.OperationName, i.Parameter,
                      ti.ClassName AS Callee,
                      ti.OperationName AS CalleeOp,
                      ti.Parameter AS CalleeOpPara
      FROM thisScen i, thisScen ti
      WHERE i.SeqNumber=reverse(right(reverse(ti.SeqNumber),
            length(ti.SeqNumber)-instr(reverse(ti.SeqNumber),'.')))
      AND   instr(ti.SeqNumber,'.')!=0
      AND i.ClassName!=ti.ClassName) fm
GROUP BY OperationName, ClassName, Parameter
ORDER BY ClassName;
```

Number of classes and methods in execution path

1. Obtain the number of classes in execution path as follows:

```
SELECT COUNT(DISTINCT class) AS numOfClass
FROM thisScen i;
```

2. Obtain the umber of methods in execution path as follows:

```
SELECT COUNT(DISTINCT *) AS numOfOp
GROUP BY OperationName, ClassName, Parameter
FROM     thisScen i;
```

Parameters for the CCP factor

To calculate the CCP factor for two classes `Ci` and `Cj`, the parameters are the number of invocations for methods (e.g. `Ci.Mi()`) originating from a class (e.g. `Cj`) and the number of invocations for methods (e.g. `Cj.Mj()`) originating from a class (e.g. `Ci`). In this case, the parameters for the query are the names of the caller and the callee (e.g. `Ci, Cj`). We can execute the following query to obtain

$$\sum Ci.fanin(Mi, Cj)$$

```
SELECT ti.Class, i.Class AS Caller,
COUNT(*) AS num
FROM thisScen i,
  (SELECT OperationName, Class, Parameter, SeqNumber
   FROM   thisScen
   WHERE  Class = 'Ci') ti
WHERE i.SeqNumber=reverse(right(reverse(ti.SeqNumber),
      length(ti.SeqNumber)-instr(reverse(ti.SeqNumber),'.')))
AND   i.Class = 'Cj'
GROUP BY i.Class;
```

7.2 Restructuring

In this section we discuss restructuring strategies.

7.2.1 Strategies to address "bad smells" in design

We can extract essential knowledge from the third-level design model to analyze and then provide results to determine the quality of the design. This process could be implemented by executing query statements. The analysis results serve as guidelines for the improvement of the design. We borrow ideas from refactoring strategies discussed in the literature such as [Fow99] [Rob] [MT], and we apply these strategies that seem to be feasible to the design model. However, we cannot address all the strategies mentioned in the literature because a number of them focus on intra-method behavior which is not captured in the UML design model. The criteria of refactoring strategies supported toward this proposal is the class-method level. In some cases the analysis results can only serve as a guidance. Developers should apply their own judgment toward whether or not to follow the suggestion provided and apply this restructuring. However, since design is not a task following a certain approach, we currently present all kinds of analysis results as suggestions, and leave the final decision to the user to decide whether or not to perform restructuring. Our approach includes the following steps: 1) Execute statements in forms of queries over the third-level design model (database), looking for "bad smells" in design, 2) Perform modifications to the second-level

design model (production system representation) following solutions from the literature mentioned above. In the following subsections we list and illustrate cases of knowledge extraction and restructuring.

Case1: Making method calls simpler with "hide method"

Motivation If a method is not used by other classes, make it private.

Strategy to obtain candidates 1) Obtain the public method list of the system,

2) Obtain those that are only used by the class within which they are defined.

General Solution Modify the visibilities of these methods to be "private."

Obtaining restructuring candidates

1. Create a view `PubOp` to obtain operations which are public with their `OperationNames`, `ClassNames`, `callers` and `SeqNumber` by the following query, whose result is captured in Table 9.

```
SELECT i.OperationName, i.Parameter,i.ClassName,
       i2.ClassName AS Caller,i.SeqNumber
FROM   interaction i, operation o, interaction i2
WHERE  o.visibility = 'public'
AND    o.OperationName=i.OperationName
AND    o.Parameter=i.Parameter
AND    o.ClassName=i.ClassName
AND    instr(i.SeqNumber,'.')!=0
AND    i2.SeqNumber=reverse(right(reverse(i.SeqNumber),
       length(i.SeqNumber)-instr(reverse(i.SeqNumber),'.')))
```

2. Create a view `PubOp2` to obtain the operation which has been used by the class other than the class in which it is defined. Then exclude these operations and obtain ones that are only used in the class in which they are defined.

OperationName	Parameter	ClassName
Op1()	NULL	Ca
Op2()	NULL	Cb
Op3()	NULL	Cc

Table 9: Result of hide method example.

```
SELECT DISTINCT p1.OperationName, p1.Parameter, p1.ClassName
FROM   PubOp p1
WHERE  p1.ClassName<>p1.Caller
 //Obtain the method called by the other
   class other than the one who defines it

SELECT DISTINCT OperationName, Parameter, ClassName
FROM  PubOp
WHERE NOT EXISTS
   (SELECT PubOp.OperationName,PubOp.ClassName,PubOp.Parameter
    FROM PubOp2
    WHERE PubOp.OperationName=PubOp2.OperationName
    AND   PubOp.ClassName=PubOp2.ClassName
    AND   PubOp.Parameter=PubOp2.Parameter)
```

Apply restructuring: As Table 9 shows, by applying the above algorithm public operations `Op1()`, `Op2()`, `Op3()` which are called only by the classes in which they are defined, are obtained . Following the solution provided, we make these methods private by modifying the production system representation as follows:

```
Before:
<Ca>::=[Operation]<Op1()>
<Ca.Op1()>::=[Visibility]<public>...
<Cb>::=[Operation]<Op2()>
<Cb.Op2()>::=[Visibility]<public>...
<Cc>::=[Operation]<Op3()>
<Cc.Op3()>::=[Visibility]<public>...
```

```
After:
<Ca>::=[Operation]<Op1()>
<Ca.Op1()>::=[Visibility]<private>...
<Cb>::=[Operation]<Op2()>
<Cb.Op2()>::=[Visibility]<private>...
<Cc>::=[Operation]<Op3()>
<Cc.Op3()>::=[Visibility]<private>...
```

Case2: Moving features between objects with "move method"

Motivation If a method is used by more features of another class than by the class in which it is defined, move that method to that class to decrease the coupling.

Strategy to find candidates 1) Obtain all operations (without constructor) together with the classes who call them 2) Obtain the classes that have the largest number of features that call an operation.

General Solution Move the method to the class with the largest number of features which use it, and make corresponding changes to the usage of this method.

Obtaining the restructuring candidates

1. By default, we deal with methods used by at least two features of a class. However, we provide the flexibility for the users to define it by themselves. These methods are potential ones that might need to be moved. We do not consider `constructor` since a constructor of a class will never be moved out of its owner class. Create a view, `Cfreq1`, which extracts the methods used by more than two features together with its `ClassName`, `Parameter`, `CallerOp`, and `Freq` (number of features)

```
SELECT OperationName,Parameter,ClassName,Caller,
COUNT(*) AS Freq
FROM (select i.OperationName, i.Parameter,i.ClassName,
             i2.ClassName AS Caller,
             i2.OperationName AS CallerOp,
             i2.Parameter AS CallerOpPara,
```

```
        COUNT(*) AS Freq
    FROM   interaction i, interaction i2
    WHERE i2.SeqNumber=reverse(right(reverse(i.SeqNumber),
          length(i.SeqNumber)-instr(reverse(i.SeqNumber),'.')))
    AND   instr(i.SeqNumber,'.')!=0
    AND i.OperationName!='constructor()'
    GROUP BY i.OperationName, i.Parameter,i.ClassName,
            Caller, i2.OperationName, i2.Parameter
    HAVING COUNT(*)>0) AS Cfreq
GROUP BY OperationName, Parameter,ClassName, Caller
HAVING COUNT(*)>2
//you can choose any number you think indicates
//a frequent usage base on the context.
```

2. The first `select` statement creates a view, `MaxFreq`, to obtain the max number of caller features. The second `select` statement creates another view, `MostF`, which obtains callers who have the most features that use the methods:

```
SELECT OperationName,Parameter,ClassName,Caller
MAX(Freq) AS MostFrequent
FROM   Cfreq1
GROUP BY OperationName, Parameter,ClassName

SELECT a1.OperationName,a1.Parameter,
       a1.ClassName, a2.Caller,
       a1.MostFrequent AS featureNum
FROM   MaxFreq a1,Cfreq1 a2
WHERE  a1.OperationName =a2.OperationName
AND    a1.Parameter=a2.Parameter
AND    a1.ClassName=a2.ClassName
AND    a2.Freq=a1.MostFrequent;
```

3. Excluding the results in which the `ClassNames` are the same as the their callers in view `MostF`.

```
SELECT *
FROM   MostF
WHERE  Caller!=ClassName;
```

OperationName	Parameter	ClassName	Caller	FeatureNum
Op1()	Cb	Ca	Cb	8

Table 10: Result of move method example.

Apply restructuring The result in Table 10 shows that `Cb` has eight features that use `Ca.Op1()`. Therefore the following modification are suggested to be applied on the production system representation:

1) Delete the following `Op1()`s method definition from Class `Ca`

```
<Ca>::=[Operation]<Op1()>
<Ca.Op1()>::=[ReturnType]<Integer>[Visibility]<public>[Parameter]<Cb>
```

2) Add method definition for `Op1()` into class `Cb`. At this step we should check whether it has a parameter of type `Cb`, in which case we delete it from the the parameter list.

```
<Cb>::=[Operation]<Op1()>
<Cb.Op1()>::=[ReturnType]<Integer>[Visibility]<public>
```

3) Modify all expressions as `Ca.Op1()` to `Cb.Op1()`

```
Before:
<Cc.Op2()>[Parameter]<int>::=[calls]<Ca.Op1()>[Parameter]<>

After:
<Cc.Op2()>[Parameter]<int>::=[calls]<Cb.Op1()>[Parameter]<>
```

Case3: Organizing data with "encapsulate collection"

Motivation if a class has collection, suggest to add `addElement` and `removeElement` operation into it.

Strategy to find candidates Obtain the classes that have collection.

General Solution Add `addElement` and `removeElement` method.

ClassName
Ca
Cb
Ch
Cj

Table 11: Result of encapsulate Collection.

Obtaining restructuring candidates

```
SELECT DISTINCT ClassName
FROM Attribute
WHERE type='set'
OR type='OrderedSet'
OR type='Bag'
OR type='Sequence';
```

Apply restructuring: The result of the query is captured in Table 11 containing the classes that have collections. Corresponding modifications should be made for the production system representation as follows: Add method definition of `addElement` and `removeElement` method for class `Ca,Cb,Ch,Cj`.

```
<Ca>::=[Operation]<addElement()>
<Ca.addElement()>::=[Visibility]<private>
<Ca>::=[Operation]<removeElement()>
<Ca.removeElemenet()>::=[Visibility]<private>
...
```

The developer might decide how to make use of methods `addElement()` and `removeElement()` to do further refactoring following the mechanism provided in [Fow99].

Case4: Dealing with generalization with "pull-up method"

Motivation If methods in sub-classes are doing the same thing, pull it up to the inheritance class.

Strategy to find candidates 1) Obtain classes list which has a parent and provide the parents' name, 2) Obtain the same operations from the classes that are of the same parent.

General Solution Change the method definition to the target class.

Obtaining restructuring candidates : Two level inheritance

1. Create a view, `extendingClassOp` to obtain the classes which have parent and their operations.

```
SELECT OperationName,Parameter,Operation.ClassName,extends
FROM  Operation,
       (SELECT c1.ClassName, c1.extends
        FROM   class c1,class c2
        WHERE  c1.extends=c2.extends
        AND    c1.ClassName<>c2.ClassName
        AND    c1.extends!=''
        AND    c2.extends!='') AS extendC
WHERE extendC.ClassName=operation.ClassName
AND   OperationName!='constructor()';
```

2. The first select statement creates a view, `SameOp`, which obtains the methods and the number of their occurrences in an inheritance relationship. The second statement obtains the common methods that are defined in subclasses.

```
SELECT OperationName, Parameter, extends AS SuperClass,
COUNT(*) AS num
FROM   extendingClassOp
GROUP BY operation, Parameter, extends
HAVING COUNT(*)=numOfChildren;

SELECT b.OperationName, b.Parameter, c.ClassName, b.SuperClass
FROM   SameOp b, extendingClassOp c
WHERE  b.OperationName=c.OperationName
AND    b.Parameter = c.Parameter
AND    b.SuperClass = c.extends
ORDER BY OperationName;
```

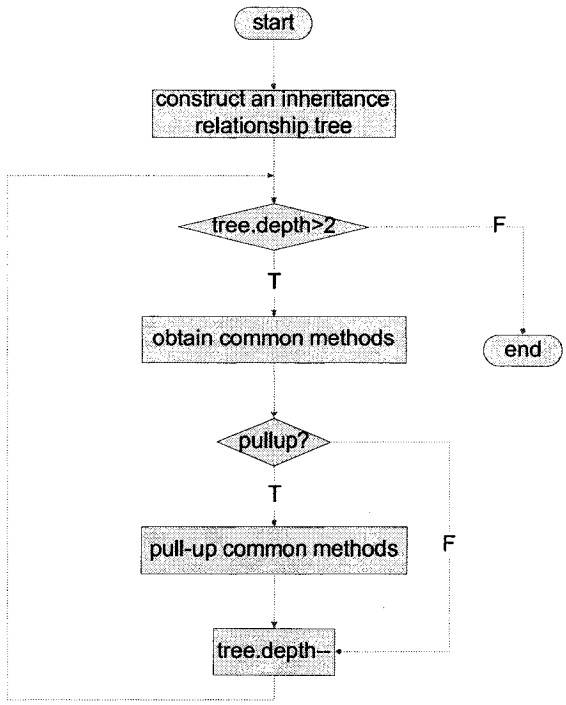

Figure 6: Algorithm illustrating multi-level inheritance.

Obtaining restructuring candidates : multi-level inheritance. Multi-level inheritance requires careful consideration. The inheritance relationship of a model can be described by a tree structure, and the restructuring candidates can be identified by recursive query over the `Class` and `Operation` tables following the idea of dealing with two level inheritance. The algorithm is described by a flowchart in Figure 6.

Figure 7 illustrates an example of a multi-level inheritance relationship starting from class `A`. According to this relationship, classes `B,C` both extend class `A`, and classes `D,E` both extend `B`, while classes `F,G` both extend `C`. `Op1()` is the common method which is a candidate to be pulled up to a higher level. The algorithm

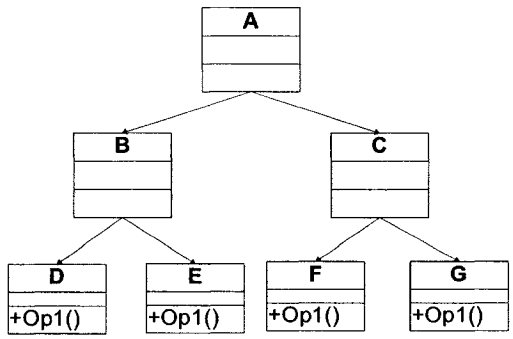

Figure 7: Multi-level inheritance.

dealing with this problem has to address inheritance relationship level by level. We can construct a tree structure in which each node stores the information that is necessary for analysis (e.g. `ClassName, parent, childrenList, methodList`). To obtain the common methods in all children of a parent, we need to traverse the tree. After obtaining `Op1()`, the common method in `D` and `E`, we first suggest to pull it up into class `B`, and perform the similarly activity on `F` and `E`. We then move to the level of class `B` and `C` in which depth we also obtain `Op1()`, the common method pulled up from the children of `B` and `C`, is the common method that can be pulled up into class `A`. Since class `A` has no parent, we stop dealing with multi-level inheritance. A suggestion is provided in each level and restructuring will be performed upon user requests.

Since not all databases have the ability of recursive query, we make use of programming language as an alternative approach. The process described by the flowchart can also be described as follows: 1) construct a tree structure by reading

OperationName	Parameter	ClassName	extends
Op5()	String	Cx	Cc
Op5()	String	Cy	Cc

Table 12: Result of pull up method example.

the classes and their corresponding properties from database (The data structure is implemented in Java in the Appendix.) 2) Get the depth of the tree structure and the nodes in each level 3) find common operations of all children of each parent in each level starting from the deepest level.

Applying restructuring: Table 12 shows the resulted restructuring candidates for pull-up method strategy, which tells that `Cx.Op5(String)` and `Cy.Op5(String)` perhaps does the same thing in the inheritance relationship among classes `Cx, Cy, Cc`. The user needs to examine the table and go to the model to determine the actual intention of method `Op5(String)` in `Cx` and `Cy`. If method `Op5()` is found to be of the same responsibility in `Cx` and `Cy`, we restructure the production system representation by pulling up the method definition to their parent class.

1. Delete the method definitions of `Op1()` and `Op2()` from `Ca` and `Cb` respectively, which means to delete the following parts:

```
<Ca>::=[Operation]<Op5()>
<Ca.Op5()>::=[Visibility]<public>
<Cb>::=[Operation]<Op5()>
<Cb.Op5()>::=[Visibility]<public>
```

2. Define the operation in their parent

```
<Cc>::=[Operation]<Op5()>
<Cc.Op5()>::=[Visibility]<public>
```

3. Modify the method call

```
Before:
<Cx.Opx()>[Parameter]<>::=[calls]<Ca.Op5()>[Parameter]<>
...
<Cy.Opy()>[Parameter]<Integer>::=[calls]<Cb.Op5()>[Parameter]<>

After:
<Cx.Opx()>[Parameter]<>::=[calls]<Cc.Op5()>[Parameter]<>
...
<Cy.Opy()>[Parameter]<Integer>::=[calls]<Cc.Op5()>[Parameter]<>
```

Case5: Dealing with generalization with "extract interface"

Motivation If several clients use the same subset of a class' interface, or two classes have a common part of their interfaces, extract this subset into an interface.

Strategy 1) Get the list of callers of all operations in a class, 2) Check whether there is a group of methods that are called by a group of clients.

General Solution If there is a group of methods that are used by a group of clients, we suggest creating an interface for the clients who use them.

Obtaining restructuring candidates

In this case, it is not necessary to analyze all classes. The potential candidates for applying "extract interface" are those which provide group of services to a group of clients. The developer should be aware of this kind of classes which take this special responsibility. Therefore we support this strategy by identifying the

OperationName	Parameter	ClassName	Caller	SeqNumber
Op1()	Integer	OriginalClass	Ca	2.1
Op1()	Integer	OriginalClass	Cb	2.1.3
Op1()	Integer	OriginalClass	Cc	3.1
Op1()	Integer	OriginalClass	Cd	4.3
Op2()	Integer	OriginalClass	Ca	2.2
Op2()	Integer	OriginalClass	Cb	3.2
Op2()	Integer	OriginalClass	Cd	3.5.1

Table 13: Result of extract interface example.

potential classes that the user would have to examine and narrowing the scope that one should examine.

1. Find all calling relationships of the user provided class.

```
SELECT DISTINCT i.OperationName, i.Parameter,i.ClassName,
       i2.ClassName AS Caller
FROM   interaction i, interaction i2
WHERE  i2.SeqNumber=reverse(right(reverse(i.SeqNumber),
       length(i.SeqNumber)-instr(reverse(i.SeqNumber),'.')))
AND instr(i.SeqNumber,'.')!=0
AND i.ClassName='UserProvidedClass'
AND i.OperationName!='constructor()'
ORDER BY OperationName;
```

Applying restructuring: Table 13 shows the restructuring candidates by providing the callers of all operations of a class.

We can see that the set of operations {`Op1()`,`Op2()`}, which is a subset of all operations in a class (here we say it is called `OriginalClass`), there is a group of clients {`Ca`,`Cb`,`Cd`} employing it. Since `Ca`,`Cb` and `Cd` use only this subset of operations, they do not need to know all responsibilities of the class. Following the guidelines of extract interface, we modify the production system representation as follows:

1. Add an interface definition as follows (the interface name should be provided by the users)

```
<ClassList>::=<...,interfaceClass>
interfaceClass::=[Class]
<interfaceClass>::=[IsInterface][Visibility]<public>
<interfaceClass>::=[Operation]<Op1()>
<interfaceClass.Op1()>::=[Visibility]<public>
<interfaceClass>::=[Operation]<Op2()>
<interfaceClass.Op2()>::=[Visibility]<public>
```

2. Modify the `[Implements]` modifier of `OriginalClass` as follows

```
<OriginalClass>::=[Visibility]<public>[Implements]<interfaceClass>
```

7.2.2 Restructuring to design patterns

With the aid of the second-level design model, patterns can be applied on design automatically upon user request. However, we can not address patterns fully in the design stage because some patterns are highly related to implementation. What we provide in this project is a set of key features for each pattern which can serve as guidelines for developers to implement a pattern. The criteria of patterns supported lies at the class-method level. Moreover, sometimes, user intervention is required when applying a particular pattern (this will be discussed in the following subsection). Further, restructuring to patterns discussed in this dissertation only addresses the static structure of the model.

Singleton: A creational pattern

Creational patterns provide guidance on how to create objects when their creation requires making decisions which need to be structured and encapsulated [Gra02].

Motivation To ensure that only one instance of a class is created.

General solution To provide a private constructor and perform lazy instantiation.

1) Private constructor: The first step is to provide a constructor that is private to prevent direct instantiation from other class. Otherwise, code generation tools might create a public constructor if none is provided.

2) Lazy instantiation: Create an accessor method to return an instance of the class itself but not to allow more than one copy to be accessed.

Example Consider a class that is defined as follows before applying the Singleton pattern.

```
<Class>::=[Property]<Attr1>
<Class.Attr1>::=[Visibility]<public>
<Class>::=[Operation]<Op1()>
<Class.Op1()>::=[ReturnType]<Integer>[Visibility]<public>
```

If developers request to apply the Singleton pattern over `Class`, we can modify the corresponding production system representation as follows:

1. Add a private constructor:

```
<Class>::=[Property]<Attr1>
<Class.Attr1>::=[Type]<Integer>[Visibility]<public>
<Class>::=[Operation]<Op1()>
<Class.Op1()>::=[ReturnType]<Integer>[Visibility]<public>
```

```
<Class>::=[Operation>[constructor()]
<Class.constructor()>::=[Visibility]<private>
```

2. Provide lazy instantiation:

   1) Define an attribute that is private, static and will later be assigned a value of null, to be named as the class name(e.g. `Class`) followed with a string "_instance_initNULL".

```
<Class>::=[Property]<Class_instance_initNULL>
<Class.Class_instance_initNULL>::=[Type]<Class>
                                  [Visibility]<private>[IsStatic]
```

   2) Define a static method normally usually called `getInstance()` with the return type of the class itself.

```
<Class>::=[Operation]<getInstance()>
<Class.getInstance()>::=[ReturnType]<Class>
                        [Visibility]<public>[IsStatic]
```

Observer: A behavioural pattern

Behavioural patterns are concerned with the interaction among objects, in a way that they communicate with each other but are loosely coupled.

Motivation Some objects (subscriber objects) need to be notified with the state changes of events of another object (publisher object) and they react in their own way when events are generated. This pattern makes the publisher interacting with the subscribers with lower coupling via interface definition. In addition, it supports the Model-View [Coa] Separation principle.

General solution Add an observer interface implemented by the subscribers. The publisher attaches the subscribers dynamically which need to be notified by the generation of an event.

Example Consider three classes are provided by users: `Publisher`, `subscriberA` and `subscriberB`. Developers ask for the deployment of the Observer pattern on them. The original definitions of these three classes are as follows:

```
<Publisher>::=[Visibility]<public>
<Publisher>::=[Property]<Attr1>
<Publisher>::=[Operation]<Op2()>
<Publisher.Op2()>::=[Visibility]<public>
<SubscriberA>::=[Visibility]<public>
...
<SubscriberB>::=[Visibility]<public>
```

To apply the Observer pattern on this system, we proceed as follows:

1. Create an `Observer` interface and make both Subscribers implement it.

   a. Add an interface called `Observer`

   ```
   <ClassList>::=<...,Observer>
   <Observer>::=<Class>
   <Observer>::=[Visibility]<public>[IsInterface]
   ```

   b. Modify the definition of class `Subscriber`

   ```
   <SubscriberA>::=[Visibility]<public>[Implements]<Observer>
   <SubscriberB>::=[Visibility]<public>[Implements]<Observer>
   ```

2. Create an interface definition of `Subject` in case that there are more than one `Publishers` in future design.

   ```
   <ClassList>::=<...,Subject>
   <Subject>::=<Class>
   <Subject>::=[Visibility]<public>[IsInterface]
   ```

3. Create a method called update() for both Subscribers and Observer respectively.

```
<SubscriberA>::=[Operation]<update()>
<SubscriberA.update()>::=[Visibility]<public>
<SubscriberB>::=[Operation]<update()>
<SubscriberB.update()>::=[Visibility]<public>
<Observer>::=[Operation]<update()>
<Observer.update()>::=[Visibility]<public>
```

4. Add a list to keep the Observers in Publisher. The list is where the Publisher knows who to inform.

```
<Publisher>::=[Property]<ObserverList>
<Publisher.ObserverList>::=[Visibility]<private>[Type]<OrderedSet>
```

5. Create three methods, called attach(), detach() and notify(), which are able to add, remove and notify an Observer, respectively. attach() adds Observers to the list, detach() removes Observers to the list. notify() notifies the Observers when events occur.

```
<Subject>::=[Operation]<attach()>
<Subject.attach()>::=[Visibility]<public>
<Subject>::=[Operation]<detach()>
<Subject.detach()>::=[Visibility]<public>
<Subject>::=[Operation]<notify()>
<Subject.notify()>::=[Visibility]<public>
```

6. Modify the class definition of Publisher to make it implement the Subject. Create method definitions to implement attach(), detach() and notify() methods in Subject.

```
<Publisher>::=[Visibility]<public>[Implements]<Subject>
<Publisher>::=[Operation]<attach()>
<Publisher.attach()>::=[Visibility]<public>
<Publisher>::=[Operation]<detach()>
<Publisher.detach()>::=[Visibility]<public>
<Publisher>::=[Operation]<notify()>
<Publisher.notify()>::=[Visibility]<public>
```

7. In case that developers want to deploy the `Observer` pattern without indicating which classes are worked on, we just create two interfaces: `Subject` and `Observer`.

Facade: A structural pattern

Structural patterns aim to describe common ways that different types of objects can be organized to work with each other [Gra02].

Motivation To simplify the usage of the system by providing an unified interface for the subsystem.

General solution Define an interface where is the single point of entry to the services of the subsystem. The Facade pattern is usually applied via the Singleton pattern through which it provides a single access point to a single instance of a class. The developers should provide essential information about 1) Which services are required to be exposed, and 2) The name of the Facade class.

Example Consider that the users want to provide an interface for services: `Service1()` from class `Ca`, `Service2()` from class `Cb`, `Service3()` from class `Cc`, which are defined as follows in production system representation:

```
<Ca>::=[Operation]<Service1()>
<Ca.Service1()>::=[Visibility]<public>[Parameter]<Integer>
<Cb>::=[Operation]<Service2()>
<Cb.Service2()>::=[ReturnType]<Integer>[Visibility]<public>
<Cc>::=[Operation]<Service3()>
<Cc.Service3()>::=[Visibility]<public>
```

Then, to apply Facade pattern we proceed as follows:

1. Define a Facade class whose name is defined by developers.

   ```
   <ClassList>::=<...,UserFacade>
   <UserFacade>::=[Class]
   <UserFacade>::=[Visibility]<public>  //A Facade class is usually public
   ```

2. Provide method definitions that offer services to the clients outside the subsystem and make the modifier and signature of a method to be the same as the service-provider defined in the subsystem.

   ```
   <UserFacade>::=[Operation]<Service1()>
   <UserFacade.Service1()>::=[Visibility]<public>[Parameter]<Integer>
   <UserFacade>::=[Operation]<Service2()>
   <UserFacade.Service2()>::=[ReturnType]<Integer>[Visibility]<public>
   <UserFacade>::=[Operation]<Service3()>
   <UserFacade.Service3()>::=[Visibility]<public>
   ```

3. Apply a Facade class with the Singleton pattern following the steps of applying the Singleton pattern.

   a. Define a private constructor for class `UserFacade` as follows

```
<UserFacade>::=[Operation]<constructor()>
<UserFacade.constructor()>::=[Visibility]<private>
```

b. Define a static method `getInstance()` which returns a type of the Facade class itself.

```
<UserFacade>::=[Operation]<getInstance()>
<UserFacade.getInstance()>::=
[ReturnType]<UserFacade>[Visibility]<public>[IsStatic]  //Singleton Pattern
```

c. Define an attribute that is private and static of type Facade class, and named as `UserFacade_instance_initNULL`

```
<UserFacade>::=[Property]<UserFacade_instance_initNULL>
<UserFacade.UserFacade_instance_initNULL>::=
[Type]<UserFacade>[Visibility]<private>[IsStatic]  //Singleton pattern
```

4. Make corresponding changes in the production system representation of interaction part related to `Service1()`, `Service2()`, `Service3()`. Since class `UserFacade` is Singleton, we do not need to keep an instance of it in the client object, which means that we do not need to define an attribute of type class `UserFacade` in class `Client`.

```
Before:
<Client.Opx()>[Parameter]<>::=[calls]<Ca.Service1()>[Parameter]<Integer>
<Client.Opy()>[Parameter]<>::=[calls]<Cb.Service2()>[Parameter]<>
<Client.Opz()>[Parameter]<>::=[calls]<Cc.Service3()>[Parameter]<>

After:
<Client.Opx()>[Parameter]<>::=[calls]<UserFacade.getInstance()>[Parameter]<>
::=[calls]<UserFacade.Service1()>[Parameter]<Integer>
::=[calls]<Cb.Service2()>[Parameter]<>
<Client.Opy()>[Parameter]<>::=[calls]<UserFacade.getInstance()>[Parameter]<>
::=[calls]<UserFacade.Service2()>[Parameter]<>
::=[calls]<Cb.Service2()>[Parameter]<>
<Client.Opz()>[Parameter]<>::=[calls]<UserFacade.getInstance()>[Parameter]<>
::=[calls]<UserFacade.Service3()>[Parameter]<>
::=[calls]<Cc.Service3()>[Parameter]<>
```

In this chapter, we discussed how various comprehension and restructuring strategies can be performed towards the design model with proper algorithms in forms of SQL statements, supported by queries examples and results.

Chapter 8

Case study: A library information system

To demonstrate our approach, we will describe a case study of an information system for a library which maintains a catalog for books and journals. Library clerks may populate the catalog by adding entries. Library clerks or library members may browse the catalog. In the following sections we focus on the main (success) scenario of the use case `Make Book Entry`.

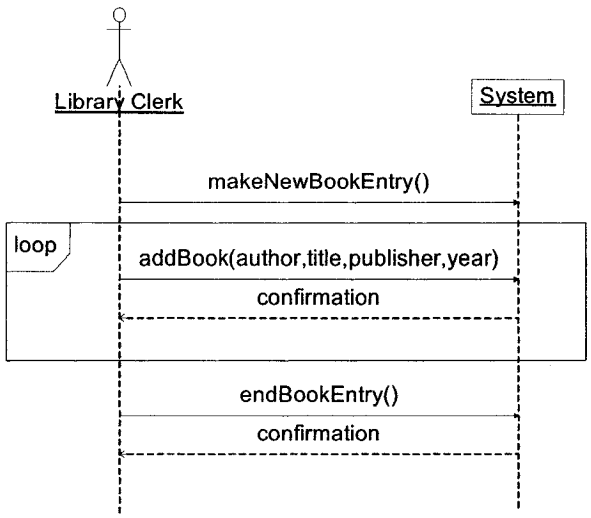

Figure 8: SSD for use case make book entry.

8.1 Design model representation

8.1.1 First-level representation: UML

The model is represented by a collection of UML artifacts. The static model is represented by the class diagram of Figure 12, and the dynamic model is represented by a collection of sequence diagrams. The use-case scenario is represented by a system sequence diagram (SSD) (Figure 8) which identifies three system operations, `makeNewBook()`, `addBook()`, `endBookEntry()`, each of which is represented by the sequence diagrams of Figures 9, 10 and 11 respectively.

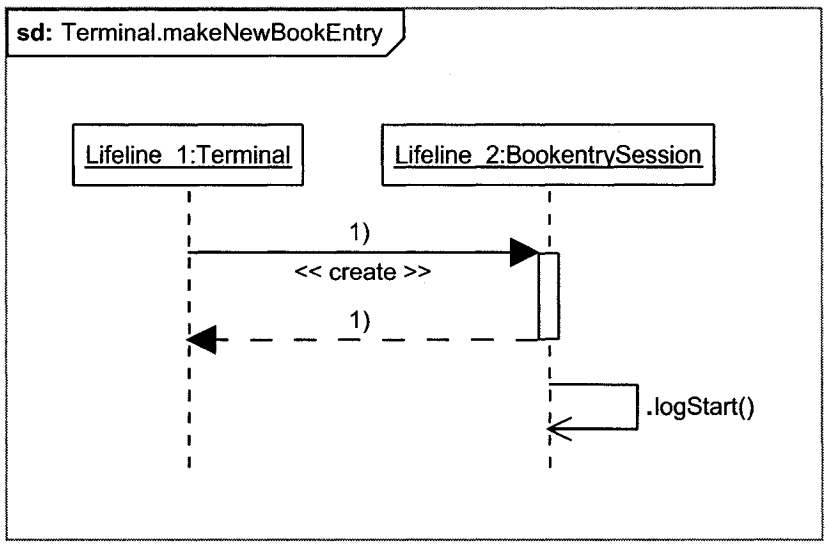

Figure 9: Sequence diagram for `makeNewBookentry()`.

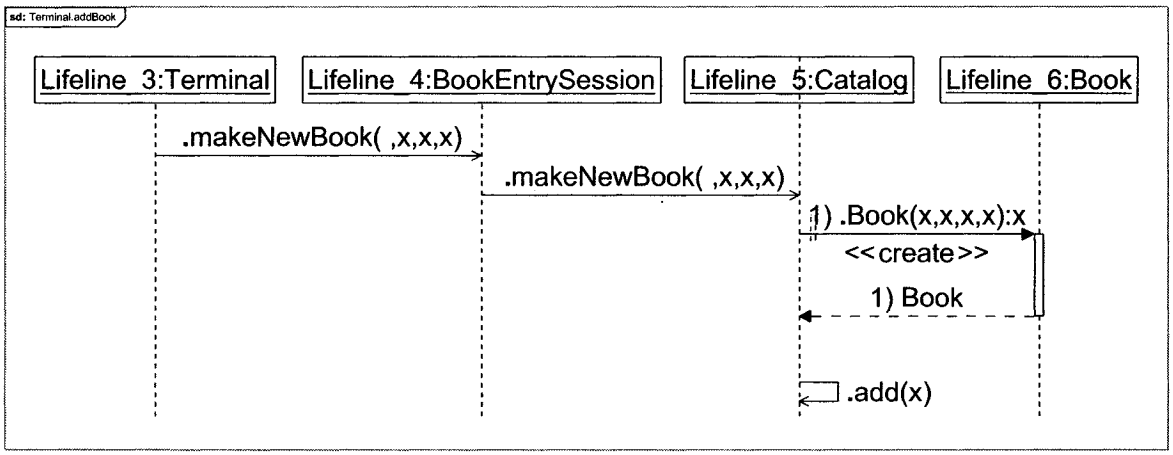

Figure 10: Sequence diagram for `addBook()`.

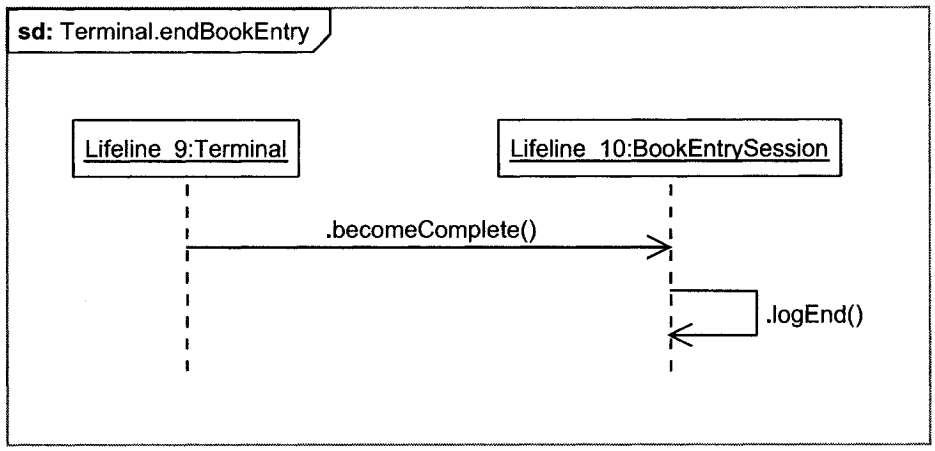

Figure 11: Sequence diagram for `endBookentry()`.

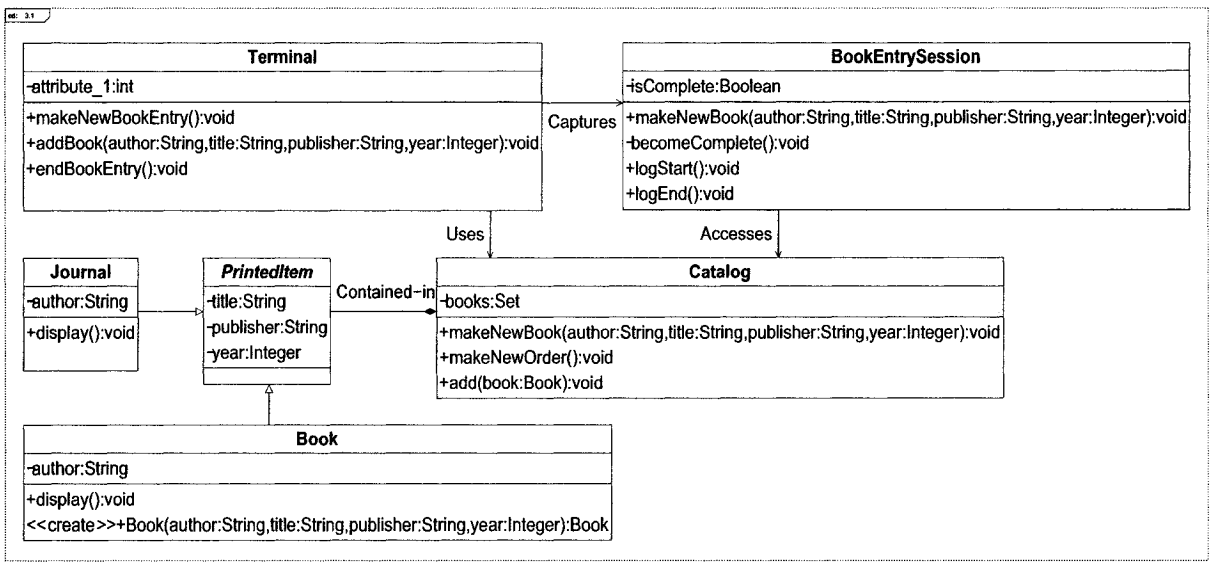

Figure 12: Class diagram.

Figure 13: Screen shot for UML2PR tool to generate the production system representation.

8.1.2 Second-level representation: Producing a production system

We translate the XMI representation of the first-level representation (UML) generated from PoseidonUML into the second-level representation (production system) by running UML2PR Ruby parser (Figure 13). The production system representation are shown below:

```
<ClassList>::=<Book,Catalog,PrintedItem,BookEntrySession,Terminal,Journal>
Book::=[Class]
Catalog::=[Class]
PrintedItem::=[Class]
BookEntrySession::=[Class]
Terminal::=[Class]
Journal::=[Class]
<Book>::=[Visibility]<public>[Extends]<PrintedItem>
<Book>::=[Property]<author>
<Book.author>::=[Type]<String>[Visibility]<private>
<Book>::=[Operation]<display()>
<Book.display()>::=[ReturnType]<void>[Visibility]<public>[Parameter]<>
<Catalog>::=[Visibility]<public>[Extends]<>
<Catalog>::=[Property]<books>
<Catalog.books>::=[Type]<Set>[Visibility]<private>
<Catalog>::=[Operation]<makeNewBook()>
<Catalog.makeNewBook()>::=[ReturnType]<void>[Visibility]<public>
                          [Parameter]<String, String, String, Integer>
<Catalog>::=[Operation]<makeNewOrder()>
<Catalog.makeNewOrder()>::=[ReturnType]<void>[Visibility]<public>[Parameter]<>
<Catalog>::=[Operation]<add()>
<Catalog.add()>::=[ReturnType]<void>[Visibility]<public>[Parameter]<Book>
<PrintedItem>::=[Visibility]<public>[Extends]<>[IsAbstract]
<PrintedItem>::=[Property]<title>
<PrintedItem.title>::=[Type]<String>[Visibility]<private>
```

```
<PrintedItem>::=[Property]<publisher>
<PrintedItem.publisher>::=[Type]<String>[Visibility]<private>
<PrintedItem>::=[Property]<year>
<PrintedItem.year>::=[Type]<Integer>[Visibility]<private>
<BookEntrySession>::=[Visibility]<public>[Extends]<>
<BookEntrySession>::=[Property]<isComplete>
<BookEntrySession.isComplete>::=[Type]<Boolean>[Visibility]<private>
<BookEntrySession>::=[Operation]<makeNewBook()>
<BookEntrySession.makeNewBook()>::=[ReturnType]<void>[Visibility]<public>
                                   [Parameter]<String, String, String, Integer>
<BookEntrySession>::=[Operation]<becomeComplete()>
<BookEntrySession.becomeComplete()>::=[ReturnType]<void>[Visibility]<private>[Parameter]<>
<BookEntrySession>::=[Operation]<logStart()>
<BookEntrySession.logStart()>::=[ReturnType]<void>[Visibility]<public>[Parameter]<>
<BookEntrySession>::=[Operation]<logEnd()>
<BookEntrySession.logEnd()>::=[ReturnType]<void>[Visibility]<public>[Parameter]<>
<Terminal>::=[Visibility]<public>[Extends]<>
<Terminal>::=[Operation]<makeNewBookEntry()>
<Terminal.makeNewBookEntry()>::=[ReturnType]<void>[Visibility]<public>[Parameter]<>
<Terminal>::=[Operation]<addBook()>
<Terminal.addBook()>::=[ReturnType]<void>[Visibility]<public>
                       [Parameter]<String, String, String, Integer>
<Terminal>::=[Operation]<endBookEntry()>
<Terminal.endBookEntry()>::=[ReturnType]<void>[Visibility]<public>[Parameter]<>
<Journal>::=[Visibility]<public>[Extends]<PrintedItem>
<Journal>::=[Property]<author>
<Journal.author>::=[Type]<String>[Visibility]<private>
<Journal>::=[Operation]<display()>
<Journal.display()>::=[ReturnType]<void>[Visibility]<public>[Parameter]<>
Begin Association
<Captures>::=[Reference1]<Terminal>[LowerBound_end1]<1>[UpperBound_end1]<1>
             [Reference2]<BookEntrySession>[LowerBound_end2]<1>[UpperBound_end2]<1>
             [Relationship]<unidirection>
<Contained-in>::=[Reference1]<Catalog>[LowerBound_end1]<1>[UpperBound_end1]<1>
                 [Reference2]<PrintedItem>[LowerBound_end2]<1>[UpperBound_end2]<1>
                 [Relationship]<composite>
<Accesses>::=[Reference1]<BookEntrySession>[LowerBound_end1]<1>[UpperBound_end1]<1>
             [Reference2]<Catalog>[LowerBound_end2]<1>[UpperBound_end2]<1>
             [Relationship]<unidirection>
<Uses>::=[Reference1]<Terminal>[LowerBound_end1]<1>[UpperBound_end1]<1>
         [Reference2]<Catalog>[LowerBound_end2]<1>[UpperBound_end2]<1>
         [Relationship]<unidirection>
End Association
Begin Interaction<Terminal.makeNewBookEntry>
<Terminal.makeNewBookEntry()>[Parameter]<>
::=[calls]<BookEntrySession.constructor()>[Parameter]<>
::=[calls]<BookEntrySession.logStart()>[Parameter]<>
End Interaction<Terminal.makeNewBookEntry>
Begin Interaction<Terminal.addBook>
<Terminal.addBook()>[Parameter]<String, String, String, Integer>
::=[calls]<BookEntrySession.makeNewBook()>[Parameter]<String, String, String, Integer>
```

Figure 14: Screen shot for PR2DB tool to build a database.

ClassName	Visibility	IsAbstract	IsInterface	Implements	IsFinal	Extends
Catalog	public	false	false	NULL	false	NULL
Journal	public	false	false	NULL	false	Printed-Item
Terminal	public	false	false	NULL	false	NULL
Book	public	false	false	NULL	false	Printed-Item
PrintedItem	public	true	false	NULL	false	NULL
BookEntrySession	public	false	false	NULL	false	NULL

Table 14: Class table.

```
::=[calls]<Catalog.makeNewBook()>[Parameter]<String, String, String, Integer>
::=[calls]<Book.constructor()>[Parameter]<String, String, String, Integer>
<Catalog.makeNewBook()>[Parameter]<String, String, String, Integer>
::=[calls]<Catalog.add()>[Parameter]<Book>
End Interaction<Terminal.addBook>
Begin Interaction<Terminal.endBookEntry>
<Terminal.endBookEntry()>[Parameter]<>
::=[calls]<BookEntrySession.becomeComplete()>[Parameter]<>
::=[calls]<BookEntrySession.logEnd()>[Parameter]<>
End Interaction<Terminal.endBookEntry>
```

8.1.3 Third-level representation: Producing a relational database schema

The second-level representation (production system) is translated into a third-level representation (relational database schema) by running PR2DB tool (Figure 14). The content of the resulted database is shown in Tables 14 - 18.

AttributeName	ClassName	Type	IsStatic	IsFinal	Visibility
books	Catalog	false	false	NULL	false
author	Journal	false	false	NULL	false
author	Book	false	false	NULL	false
title	PrintedItem	false	false	NULL	false
publisher	PrintedItem	true	false	NULL	false
year	PrintedItem	true	false	NULL	false
isComplete	BookEntrySession	false	false	NULL	false

Table 15: Attribute table.

Operation-Name	Return-Type	Is-Abstract	ClassName	Is-Static	Is-Final	Visi-bility	Parameter
constructor()	void	false	Book	false	false	public	String,String, String,Integer
display()	void	false	Book	false	false	public	NULL
makeNewBook()	void	false	Catalog	false	false	public	String,String, String,Integer
makeNewOrder()	void	false	Catalog	false	false	public	NULL
add()	void	false	Catalog	false	false	public	Book
makeNewBook()	void	true	BookEntry-Session	false	false	public	String,String, String,Integer
become-Complete()	void	false	BookEntry-Session	false	false	private	NULL
logStart()	void	false	BookEntry-Session	false	false	public	NULL
logEnd()	void	false	BookEntry-Session	false	false	public	NULL
makeNewBook-Entry()	void	false	Terminal	false	false	public	NULL
addBook()	void	false	Terminal	false	false	public	String,String, String,Integer
addBookEntry()	void	false	Terminal	false	false	public	NULL
display()	void	false	Journal	false	false	public	NULL

Table 16: Operation table.

Assoc-iation-Name	end1	end2	Lower-Bound_end1	Upper-Bound_end1	Lower-Bound_end2	Upper-Bound_end2	Rela-tion-ship
Capture	Terminal	BookEntry-Session	1	1	1	1	uni-direction
Accesses	Catalog	PrintedItem	1	1	1	1	composite
Con-tained-in	BookEntry-Session	Catalog	1	1	1	1	uni-direction
Uses	Terminal	Catalog	1	1	1	1	uni-direction

Table 17: Association table.

OperationName	ClassName	Interaction-Name	SeqNumber	Parameter	Looping
makeNew-BookEntry()	Terminal	Terminal.make-NewBookEntry	1	NULL	0
constructor()	BookEntry-Session	Terminal.make-NewBookEntry	1.1	NULL	0
logStart()	BookEntry-Session	Terminal.make-NewBookEntry	1.1.1	NULL	0
addBook()	Terminal	Terminal.add-Book	2	String,String, String,Integer	0
makeNewBook()	BookEntry-Session	Terminal.add-Book	2.1	String,String, String,Integer	0
makeNewBook()	Catalog	Terminal.add-Book	2.1.1	String,String, String,Integer	0
constructor()	Book	Terminal.add-Book	2.1.1.1	String,String, String,Integer	0
add()	Catalog	Terminal.add-Book	2.1.1.2	Book	0
endBookEntry()	Terminal	Terminal.end-BookEntry	3	NULL	0
becomeComplete()	BookEntry-Session	Terminal.end-BookEntry	3.1	NULL	0
logEnd()	BookEntry-Session	Terminal.end-BookEntry	3.1.1	NULL	0

Table 18: Interaction table.

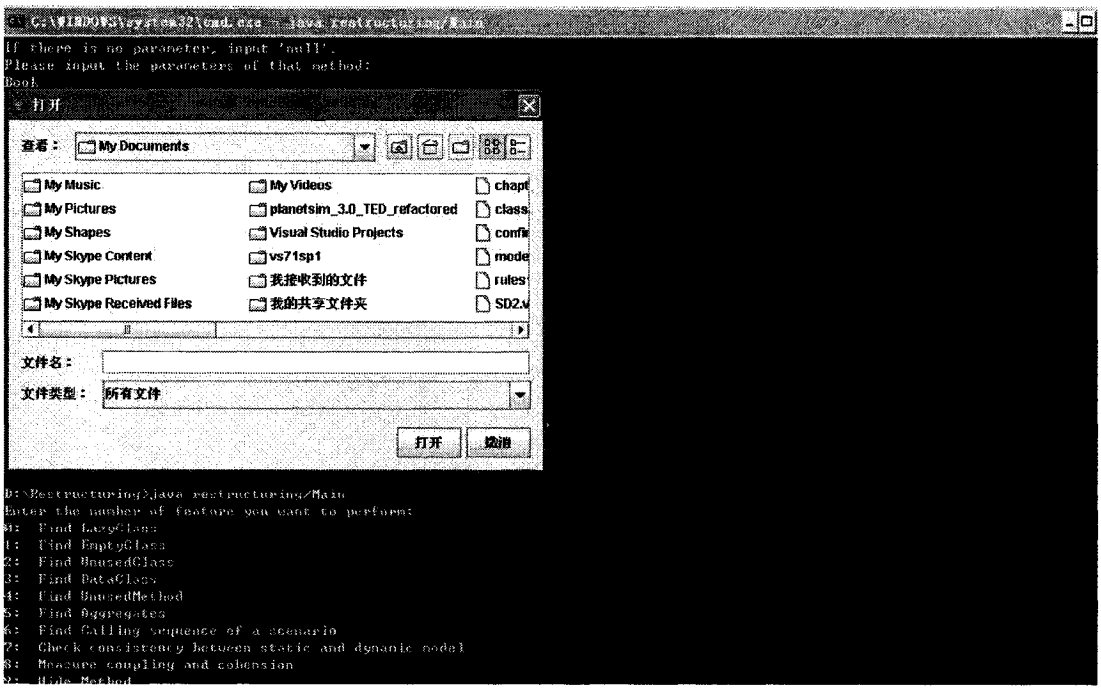

Figure 15: Screen shot of RPR tool to perform model comprehension.

UnusedMethod	ClassName	Parameter
display()	Book	NULL
makeNewOrder()	Catalog	NULL
display()	Journal	NULL

Table 19: Result of UnusedMethod.

8.2 Comprehension of the model

We analyze the model with the restructuring tool RPR to gain comprehension (Figure 15). RPR provides a list of features which could be applied to perform various types of analyzing strategies. The analyzing results are provided in forms of tables.

1. Perform "Find UnusedMethod" and obtain Table 19 as a result.

ClassName
PrintedItem

Table 20: Result of DataClass.

MethodName	ClassName	Parameter
makeNewBookEntry()	Terminal	NULL
constructor()	BookEntrySession	NULL
logStart()	BookEntrySession	NULL
addBook()	Terminal	String,String, String,Integer
makeNewBook()	BookEntrySession	String,String, String,Integer
makeNewbook()	Catalog	String,String, String,Integer
constructor()	Book	String,String, String,Integer
add()	Catalog	Book
endBookEntry()	Terminal	NULL
becomeComplete()	BookEntrySession	NULL
logEnd()	BookEntrySession	NULL

Table 21: Result of calling sequence.

2. Perform "Find DataClass" and obtain Table 20 as a result.

3. Perform "Find Calling sequence of a scenario" by being provided system operations in terms of the names of their corresponding sequence diagrams, in the following order: `Terminal.makeNewBookEntry, Terminal.addBook, Terminal.endBookEntry`. The result is captured in Table 21.

4. Perform "Check consistency between static and dynamic model". The result is captured in Table 22.

MethodName	OwnerClass	Parameter	Caller
becomeComplete()	BookEntrySession	NULL	Terminal

Table 22: Result of checking consistency.

Figure 16: Screen shot for RPR tool to perform model restructuring.

MethodName	ClassName	Parameter
add()	Catalog	Book
logStart()	BookEntrySession	NULL
logEnd()	BookEntrySession	NULL

Table 23: result of HideMethod.

8.3 Performing Restructuring

The RPR tool also provides functionalities to restructuring the model. Various kinds of strategies are provided (Figure 16):

1. Perform "Hide Method" and obtain the result captured in Table 23.

   After obtaining the restructuring candidates, restructuring is performed upon user decision. In this case, the user decides to make all of them private, The RPR tool will perform automatic modification over the production system representation as follows:

```
Before:
<Catalog>::=[Operation]<add()>
<Catalog.add()>::=[ReturnType]<void>[Visibility]<public>[Parameter]<Book>
...
<BookEntrySession>::=[Operation]<logStart()>
<BookEntrySession.logStart()>::=[ReturnType]<void>[Visibility]<public>[Parameter]<>
<BookEntrySession>::=[Operation]<logEnd()>
<BookEntrySession.logEnd()>::=[ReturnType]<void>[Visibility]<public>[Parameter]<>

After:
<Catalog>::=[Operation]<add()>
<Catalog.add()>::=[ReturnType]<void>[Visibility]<private>[Parameter]<Book>
```

```
...
<BookEntrySession>::=[Operation]<logStart()>
<BookEntrySession.logStart()>::=[ReturnType]<void>[Visibility]<private>[Parameter]<>
<BookEntrySession>::=[Operation]<logEnd()>
<BookEntrySession.logEnd()>::=[ReturnType]<void>[Visibility]<private>[Parameter]<>
```

2. Perform "Pull-up method". The result is captured in Table 24. In this case, method `display()` in both `Book` and `Journal` is decided to be pulled up to `PrintedItem`. Automatic modification is performed over the production system representation as follows:

```
Before:
...
<Book>::=[Operation]<display()>
<Book.display()>::=[ReturnType]<void>[Visibility]<public>[Parameter]<>
<PrintedItem>::=[Visibility]<public>[Extends]<>[IsAbstract]
<PrintedItem>::=[Property]<title>
<PrintedItem.title>::=[Type]<String>[Visibility]<private>
<PrintedItem>::=[Property]<publisher>
<PrintedItem.publisher>::=[Type]<String>[Visibility]<private>
<PrintedItem>::=[Property]<year>
<PrintedItem.year>::=[Type]<Integer>[Visibility]<private>
<Journal>::=[Operation]<display()>
<Journal.display()>::=[ReturnType]<void>[Visibility]<public>[Parameter]<>
...

After:
...
<PrintedItem>::=[Visibility]<public>[Extends]<>[IsAbstract]
<PrintedItem>::=[Operation]<display()>
<PrintedItem.display()>::=[ReturnType]<void>[Visibility]<public>[Parameter]<>
<PrintedItem>::=[Operation]<display()>
<PrintedItem.display()>::=[ReturnType]<void>[Visibility]<public>[Parameter]<>
<PrintedItem>::=[Property]<title>
<PrintedItem.title>::=[Type]<String>[Visibility]<private>
<PrintedItem>::=[Property]<publisher>
<PrintedItem.publisher>::=[Type]<String>[Visibility]<private>
<PrintedItem>::=[Property]<year>
<PrintedItem.year>::=[Type]<Integer>[Visibility]<private>
...
```

MethodName	Parameter	className	Parent
display()	NULL	Book	PrintedItem
display()	NULL	Journal	PrintedItem

Table 24: Result of Pull up method.

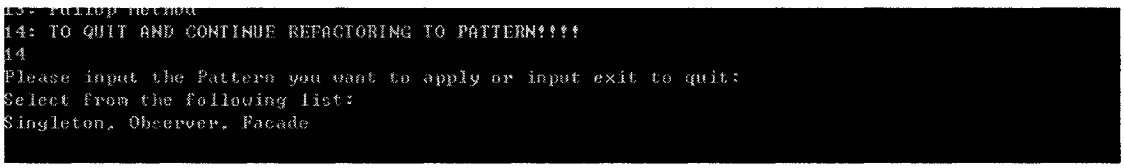

Figure 17: Screen shot for RPR tool to perform model restructuring to patterns.

3. The RPR tool supports "Restructuring to patterns" (Figure 17).

   The Singleton design pattern is applied over class `Catalog` upon user request, since there should be only one entrance for object `Catalog`. Key features of the Singleton pattern will be added on the corresponding part of the production system representation as follows:

```
Before:
<Catalog>::=[Visibility]<public>[Extends]<>
<Catalog>::=[Property]<books>
<Catalog.books>::=[Type]<Set>[Visibility]<private>
<Catalog>::=[Operation]<makeNewBook()>
<Catalog.makeNewBook()>::=[ReturnType]<void>[Visibility]<public>
                          [Parameter]<String,String,String,Integer>
<Catalog>::=[Operation]<makeNewOrder()>
<Catalog.makeNewOrder()>::=[ReturnType]<void>[Visibility]<public>[Parameter]<>
<Catalog>::=[Operation]<add()>
<Catalog.add()>::=[ReturnType]<void>[Visibility]<public>[Parameter]<Book>

After:
<Catalog>::=[Visibility]<public>[Extends]<>
<Catalog>::=[Operation]<constructor()>
<Catalog.constructor()>::=[Visibility]<Private>
<Catalog>::=[Property]<Catalog\_instance\_initNULL>
<Catalog.Catalog\_instance\_initNULL>::=[Type]<Catalog>[Visibility]<Private>[IsStatic]
<Catalog>::=[Operation]<getInstance()>
<Catalog.getInstance()>::=[ReturnType]<Catalog>[Visibility]<Public>[IsStatic]
<Catalog>::=[Property]<books>
```

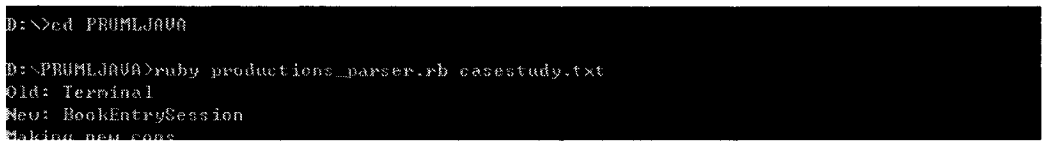

Figure 18: Screen shot for PR2JAVA tool to generate Java skeletal code.

```
<Catalog.books>::=[Type]<Set>[Visibility]<private>
<Catalog>::=[Operation]<makeNewBook()>
<Catalog.makeNewBook()>::=[ReturnType]<void>[Visibility]<public>
                          [Parameter]<String,String,String,Integer>
<Catalog>::=[Operation]<makeNewOrder()>
<Catalog.makeNewOrder()>::=[ReturnType]<void>[Visibility]<public>[Parameter]<>
<Catalog>::=[Operation]<add()>
<Catalog.add()>::=[ReturnType]<void>[Visibility]<public>[Parameter]<Book>
```

8.4 Producing a refined first-level model representation

To obtain a refined UML model, a set of Java skeletal code is generated by the PR2UML tool (Figure 18, 19). We import these `.java` file into EclipseUML and obtain a new UML model (Figure 20).

In this chapter, we provided a case study to demonstrate how to obtain comprehension and to perform restructuring of a software system, adopting the three-level design model, together with our comprehension and restructuring strategies.

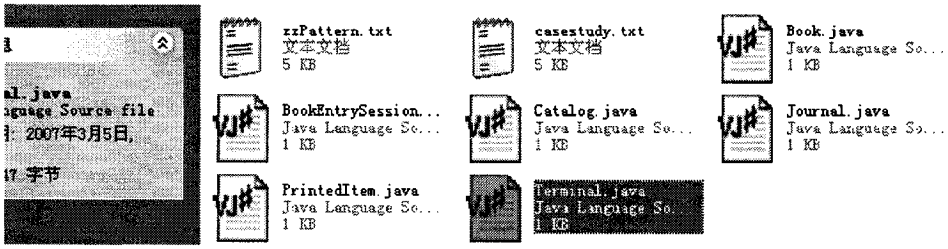

Figure 19: Screen shot illustrating *.java files generated.

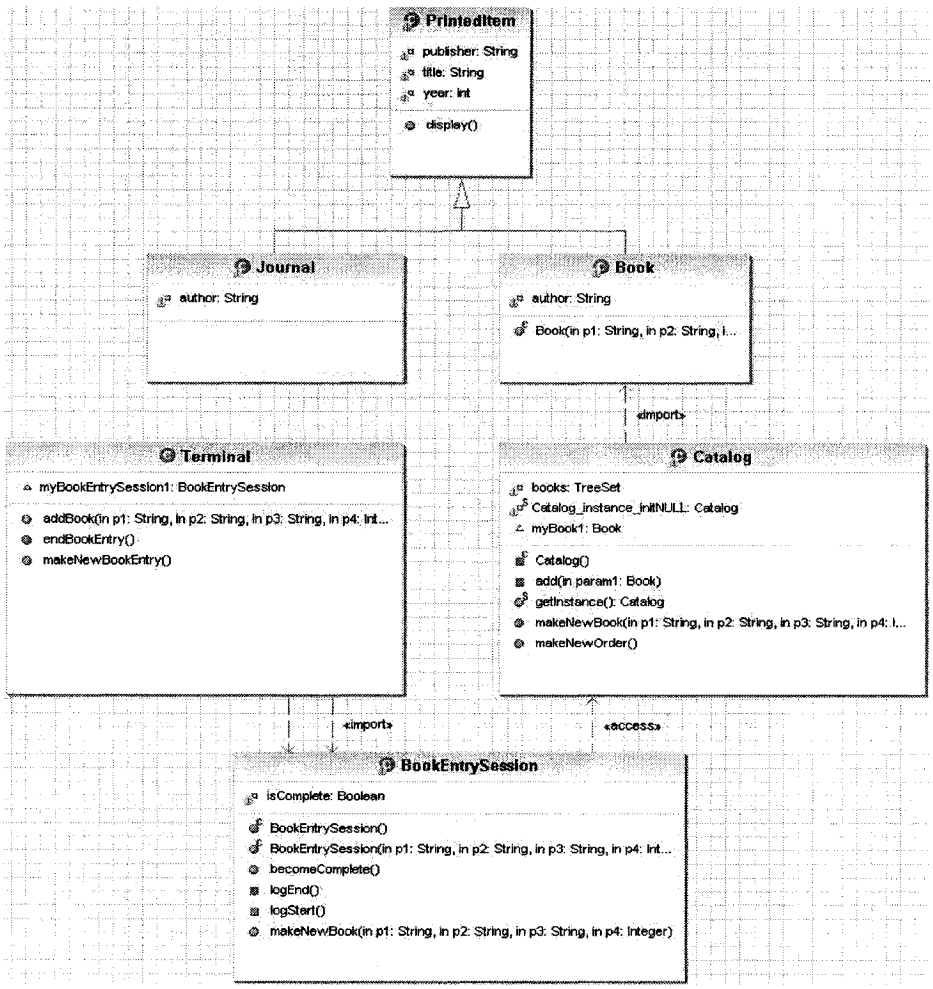

Figure 20: Refined class diagram.

Chapter 9

Related work and evaluation

Model transformation has widely been discussed in the literature and there is currently a number of tools available. MTF [MTF] is a commercial tool from IBM which supports transformation between EMF models. In addition, refactoring has been discussed in [Fow99]. Supporting tools to facilitate refactoring process are available currently. Eclipse provides help for refactoring the code, which restructures the code only but needs decision and input from the developers to what and where it should target. codePro Analytix [cod] is a testing tool to analyze and address problems over Java code. Metrics [met] for Eclipse is a plug-in for Eclipse and it provides metrics for analyzing Java code. Borland Together is a commercial tool providing audit and metrics for detecting bad smells in code. MagicDraw is a commercial tool that provides metrics for both UML model and code, performing simple statistics toward the model elements without providing suggestion on

restructuring candidates and automatic restructuring. A survey from [MT] shows that the research focusing on refactoring in the design stage gains great emphasis. Refactoring on UML model has been discussed in [SPTJ] [Ast]. In these, the authors proposed the restructuring of the UML model by identifying the restructuring candidates through completely manual analysis based on certain guidelines. Refactoring to patterns with supporting tools has been discussed in [Arn], where the author provides users with a "pattern library", a set of components capturing the intent of the underlying design patterns that they can reuse directly. Moreover in [FBY], the authors provide approaches and tool support that can generate pattern-prescribed code automatically. In [DYZ], the authors provide approaches and tool support for transforming a UML model of a design pattern into the evolved UML mode of the pattern. In [GSMD] and [BPPT], the authors discuss how to keep consistency among different UML models and code after applying refactoring on one UML model.

In [MDB+] the authors pose the following question as future work: How can we apply refactoring at higher levels of abstraction? The recent released version of ArgoUML, a commercial tool, starts to provide the functionality of providing suggestions in forms of a "to-do list" based on a design analysis. However, according to its user manual [RVR+], each suggestion it provides is mainly based on separated analysis of a concrete type of diagram (e.g. analysis on class diagram, analysis on state diagram) and the analysis is still in a simple level (e.g. naming

problem). Rational Software Architect (RSA) [BIJ] [RSA], a popular commercial tool for Model Driven Development (MDD), supports application of patterns via templates in design level by necessary user intervention in order to provide template parameters. Nevertheless, it does not support comprehension and also does not provide restructuring candidates for model restructuring.

Our three level design model architecture and supporting tools, can automate the model transformation process and can provide assistance with comprehension and restructuring of the system. This is achieved by analyzing the static, and dynamic design model in both separated and combined way (e.g.checking consistency between static and dynamic model) to gain comprehension, obtain restructuring candidates and provide automatic restructuring upon the user determination or request. This implies that, between a UML design model and its implementation (neglecting the way it is achieved), we add another layer composed by the second, and the third level design model to gain more confidence over the design and to improve it, if necessary. Thus, along with MDA we build a more reliable design model that is platform-independent. This way, regardless of the specific technology a design is implemented, the corresponding generated code can be of higher quality.

Comprehension and transformation can be tedious under some other technologies and approaches. For example, using imperative language (e.g.C++), to produce the production system representation, or even working with XMI representation, can implement the same logic but it usually requires more code to parse the file. For example, a query to find out all public classes requires parsing the whole file line by line, selecting the desired data, storing them in the memory, and analyze them. However by transforming the text into a database, obtaining all public classes can be achieved by a simple query toward the class table. We can also use XSLT [W3C07] allows defining rules or templates to analyze XML file and transform XML to get the desired result. However, XSLT is also based on XML and the rule or template to transform XML file is also XML style. Therefore, it also complicate the implementation of logic. To compare with XSLT, SQL is more expressive and it is more closer to natural language. Those are the main advantages of adopting database as the third-level model in our system. The database is designed to be the analysis database rather than the transaction database. This database is good to extract knowledge from the system but it does not provide viable representation for restructuring. Therefore we do not modify the model through modifying the production system representation rather than through modifying the database.

Moreover, we define our own data types for the second and the third design model. The data types are defined to be the mathematical ones to highlight the deployment of platform-independent modeling. Currently, UML modeling tools

provide language specific data types for the designers to choose. However, since the purpose of utilizing UML is to build platform-independent models, we felt it is much better to adopt typical and general data types based on OCL, and leave the generation of language specific data types to the code generator.

The tool for analyzing the database and restructuring the production system representation can be easily extended to support new design patterns and refactoring strategies, since each functionality is well encapsulated. For example, to add a new refactoring strategy, one just needs to write the corresponding SQL statements which implement the logic of the strategy, and then add it with JDBC processing into our restructuring tool.

Chapter 10

Conclusion and recommendations

In this dissertation, we defined a three-level design model composed of a UML representation, a production system representation and a database representation, in order to facilitate the comprehension and restructuring of the object-oriented model. In order to achieve the desired results, a set of production rules has been defined, based on which the production system representation (the second-level design model) can be generated. A relational database schema has been characterized to store the production system representation for constructing the third-level design model. Based on the three level design model, we also built a set of comprehension and restructuring strategies based on certain guidelines. These guidelines are based on related strategies from the literature in order to audit and improve code. Moreover, to provide automation for model transformation, we developed a

tool suite, containing UML2PR, PR2DB, RPR and PR2UML, which can be deployed to be mutually supportive with visualization tools such as Poseidon and EclipseUML.

As a future development, this project could be extended to support the application of more complex design patterns and more refactoring strategies, by defining the algorithms to analyze the database and also by extending the restructuring tool developed. Meanwhile, the tools for different level of model transformation can be integrated to provide more facilities to the developers. Facilities for bi-directional transformation and synchronization between the second- and third-level design model could be developed for the best utilization of each level of model and keeping consistency between them. Furthermore, expert system technology could be adopted in the future to help developers make decisions after obtaining restructuring candidates in order to reduce user intervention which is required in our system.

Even if tools which support UML claim to use it to build platform-independent model, they do not work in an absolutely platform-independent way (e.g. usage of data type of a specific programming language). Therefore, to make them completely platform-independent, the data types which are allowed in the UML model should be only typical and general ones (e.g. `OrderedSet` rather than `Arraylist` in Java).Code generators should be powerful enough to identify these general data types and provide mapping to language-specific data types (if it is one-to-many

mapping, it is left to the developer to decide which one to choose). In addition, since current utilization of XMI, as an intermediate for model transformation, can cause incompatibilities among different modeling tools [IBM], in the future we expect modeling tools to follow a uniform version of XMI in order to make models generated from different tools compatible with each other. Different vendors should support this standard in order to share model information among each other. Tools developed in this dissertation can only parse the XMI representation that is generated from Poseidon and MagicDraw. We expect this situation to improve in the future with the adoption of a standard by commercial vendors, thus allowing a full integration of our tool suite with any visualization tool. Furthermore, round-trip in dynamic view from code to UML needs a better support from the current commercial visualization tools (e.g. to generate UML interaction diagrams from source code). Currently, there are few tools providing this facility such as Borland Together, MagicDraw and EclipseUML.

Appendix A

Tool support and user manual

This appendix serves as a manual on how to deploy the supporting tools to this research.

The supporting tools we provide include four parts: 1) UML2PR: XMI-to-PR parser 2) PR2DB: PR-to-database parser 3) RPR: Model analyzing and restructuring tool 4) PR2UML: PR-to-Java skeletal code parser. The current version of XMI-to-PR parser is able to deal with XMI file generated from Poseidon UML.

1. Create UML diagrams (class diagram and sequence diagrams) in Poseidon and generate XMI file of the project. According to our requirement, when defining data types for attributes and return types of methods, what a designer is allowed to enter is defined data types in production system. If other types are used in the model, it can not generate correct Java skeletal code. In addition, in order to have the right order of events, the designer should

create the sequence diagrams following the order of system operations of each scenario since this is the way we get the right calling sequence for scenarios.

2. To parse the XMI file, copy the XMI file into the directory in which you have the XMI parser UML2PR. The parser can be invoked by going to the directory where you put the parser together with the .xmi file and running command `ruby xml_parser.rb input.xmi output.txt` ("input" is the name of the .xmi file, "output" is the name of the production system representation file in the form of .txt file).

3. To transfer the data of production system representation into database, run PR2DB tool. To invoke this tool, go to the directory where you put the tool together with the production system representation file generated from UML2PR tool and execute command `java PRmySQL/PR`).

4. To analyze the database and to restructure the production system representation, invoke the RPR tool by going to the directory where you put the RPR and running command `java restructuring/Main`. Then follow the instructions provided by the application to perform the desired strategies.

5. To transform the restructured production system representation to java skeletal code, run PR2JAVA tool. The tool can be invoked by going to the directory where you put the ruby productions_parser and executing command

`ruby production_parser.rb input.txt` ("input" is the name of the production system representation file based on which you want to generated the .java file). Subsequently, you include these files in an eclipse project (which includes the eclipseUML plug-in from Omondo), creating class diagrams for the whole project and creating sequence diagrams for the desired methods.

Appendix B

Addressing multi-level inheritance

The following Java implementation addresses multi-level inheritance and it is being deployed by our tools.

```
import java.util.ArrayList;
public class Node {
  public Node parent = null;
  public ArrayList childrenList = new ArrayList();
  public String ClassName="";
  public ArrayList methodList = new ArrayList();
  private boolean isRoot;
  public Node() { }
  public void addChild(Node child) {
    childrenList.add(child);
  }
  public Node getChild(int index) {
    return (Node) childrenList.get(index);
  }
  public int getChildCount() {
    return childrenList.size();
  }
  public int getDepth() {
    if (isRoot())
      return 1;
    else
      return 1 + getParent().getDepth();
```

```
  }
  public boolean isInternalNode() {
    if (childrenList != null)
      return true;
    else
      return false;
  }
  public boolean isLeaf() {
    if (childrenList == null)
      return true;
    else
      return false;
  }
  public boolean isRoot() {
    return isRoot;
  }
  public void removeChildAt(int index) {
    childrenList.remove(index);
  }
  public void setRoot() {
    isRoot = true;
  }
  public Node getParent() {
    return parent;
  }
  public void setParent(Node parent) {
    this.parent = parent;
  }
  public ArrayList getChildrenList() {
    return childrenList;
  }
  public void addMethod(Method method) {
    methodList.add(method);
  }
}
```

Appendix C

Glossary and abbreviations

Software Comprehension A process of obtaining knowledge from a software system.

Bad smells Factors that will affect the quality of the software system originally discussed in the context of code.

Design pattern A description of communicating objects and classes that are customized to solve a general design problem in a particular context.

Extensible Markup Language (XML) A general-purpose markup language that supports various applications and is recommended by W3C.

EMF Eclipse Modeling Framework.

Data class A class contains only fields without methods.

Lazy class A class with few responsibilities.

Model Driven Architecture (MDA) A software design approach recommended by Object Management Group (OMG) which supports *Model Driven Engineering.*

Model Driven Engineering (MDE) A software development approach that uses models as primary engineering artifacts throughout the engineering lifecycle.

MTF Model Transformation Framework.

Model transformation A process of transforming a model based on a metamodel, into another model based on either the same metamodel or a different metamodel.

Platform-independent model (PIM) A model of a software system that is independent of the specific implementation technology.

Platform-specific model (PSM) A model of a software system that is linked to a specific implementation technology.

PR2DB Production system representation to database schema tool.

PR2UML Production system representation to *UML* tool.

Production Rules A set of abstract rules to describe semantics of a *UML* model.

PRP Restructuring of production system representation tool.

RSA Rational Software Architect.

System Sequence Diagram (SSD) A picture that describes the events that external actors generate, their order and possible inter-system events, for a use-case scenario.

Unified Modeling Language (UML) A general purpose graphical modeling language for object-oriented modeling.

UML2PR UML to production system representation tool.

W3C World Wide Web Consortium.

XML Schema Definition (XSD) A recommendation of *W3C* which specifies how to formally describe the elements in an *XML* document.

XML Metadata Interchange (XMI) A standard for exchanging metadata information using *XML*.

Bibliography

[Arn] Karine Arnout. From Patterns to Components. Ph.D Thesis, Swiss Institute of Technology, March 31,2004.

[Ast] Dave Astels. Refactoring with UML. Proceedings of 3rd International Conference on eXtreme Programming and Flexible Processes in Software Engineering, pp. 67–70, Alghero, Sardinia, Italy. 26th-30th May, 2002.

[BIJ] A. W. Brown, S. Iyengar, and S. Johnston. A Rational Approach to Model-Driven Development. IBM System Journal, Volume 45, Number 3, pages 463-480, 2006.

[BPM] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program Transformations for Practical Scalable Software Evolution. Proceedings of 26th International Conference on Software Engineering, Scotland, UK, May, 2004.

[BPPT] Paolo Bottoni, Francesco Parisi-Presicce, and Gabriele Taentzer. Coordinated Distributed Diagram Transformation for Software Evolution. Electronic Notes in Theoretical Computer Science. Volume: 72, Issue: 4. 2002.

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. The Unified Modeling Language User Guide, 1999.

[BSM+] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework.* Addison Wesley. pages 93-114. 2004.

[CH] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. Proceedings OOPSLA03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003.

[Coa] Peter Coad. Object-oriented patterns. *Communications of the ACM.* Volume 35, Number 9, pages 152-159. 1992.

[cod] CodePro Analytix from Instantiations. `http://www.instantiations.com/codepro/index.html`. Date last accessed: 6th March, 2007.

[DYZ] Jing Dong, Sheng Yang, and Kang Zhang. A Model Transformation Approach for Design Pattern Evolutions. Proceedings of 13th Annual IEEE International Conference on Engineering of Computer Based Systems (ECBS), pages 80-89, Germany, March, 2006.

[Ecl] EclipseUML Free Edition. `http://www.omondo.com`. Date last accessed: 5th March, 2007.

[EMF] Eclipse Modeling Framework. `http://www.eclipse.org/modeling/emf/?project=emf`. Date last accessed: 5th March, 2007.

[FBY] John Vlissides Frank Budinsky, Marilyn Finnie and Patsy Yu. Automatic Code Generation From Design Patterns. IBM System Journal. Volume: 35, Number: 2, Pages: 151 - 171. 1996.

[FCa] Amir Abdollahi Foumani and Constantinos Constantinides. Aspect-oriented Reverse Engineering. Proceedings of the 9th World Multi-conference on Systemics, Cybernetics and Informatics (WMSCI 2005), Orlando, Florida, USA, 10th-13th July, 2005.

[FCb] Amir Abdollahi Foumani and Constantinos Constantinides. Reengineering Object-oriented Artifacts by Analyzing Dependency Graphs and Production Rules. Proceedings of the 9th IASTED International Conference on Software Engineering and Applications (SEA 2005). pp.335-343. Phoenix, AZ, USA, 14th-16th Novermber, 2005.

[Fow99] Martin Fowler. *Refactoring-Improving the Design of Existing Code.* Addison-Wesley, 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns-Elements of Reusable Object-Oriented Software: Professional Computing Series.* Addison Wesley, 1995.

[Gra02] Mark Grand. *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML.* Wiley, second edition, 2002.

[Groa] Object Management Group. The Model-Driven Architecture. The Model-Driven Architecture, Guide Version 1.0.1. 6th January, 2003.

[Grob] Object Management Group. MOF 2.0/XMI Mapping Specification, Version 2.1. 1st September, 2005.

[GSMD] Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Formal UML Support for the Semi-automatic Application of Object- Oriented Refactorings. Technical Report, University of Antwerp, 2003.

[IBM] Working XML: UML, XMI, and Code Generation, Part 2. `http://www-128.ibm.com/developerworks/library/x-wxxm24/`. Date last accessed: 5th March, 2007.

[Ker04] Joshua Kerievsky. *Refactoring to Patterns.* Addison Wesley, 2004.

[Lar04] Craig Larman. *Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design and Iterative Development.* Addison-Wesley, 2004.

[MDB+] Tom Mens, Serge Demeyer, Bart Du Bois, Hans Stenten, and Pieter Van Gorp. Refactoring: Current Research and Future Trends. ETAPS 2003 Workshop on Language Descriptions, Tools and Applications. Warsaw, Poland, 5th-13th April, 2003.

[met] Metrics 1.3.6 - Getting started. `http://metrics.sourceforge.net`. Date last accessed: 15th March, 2007.

[MT] Tom Mens and Tom Tourwe. A Survey of Software Refactoring. *IEEE Trans. on Software Engineering.* Volume: 30, Issue: 2, pp.126-139. February, 2004.

[MTF] AlphaWorks: Model Transformation Framework: Overview. `http://www.alphaworks.ibm.com/tech/mtf`. Date last accessed: 5th March, 2007.

[PFC] Paria Parsamanesh, Amir Abdollahi Foumani, and Constantinos Constantinides. Mining Anomalies in Object-Oriented Implementations Through Execution Traces. Proceedings of the International Conference on Software and Data Technologies (ICSOFT), Setubal, Portugal. 11th-14th September 2006.

[Pos] Gentleware - Model to Business: UML Tools and Services. `http://www.gentleware.com`. Date last accessed: 5th March, 2007.

[Rob] Donald Roberts. Practical Analysis for Refactoring. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[RSA] Rational Software Architect - Features and Benefits. `http://www-306.ibm.com/software/awdtools/architect/swarchitect/features/index.html?S_CMP=rnav`. Date last accessed: 15th March, 2007.

[RVR+] Alejandro Ramirez, Philippe Vanpeperstraete, Andreas Rueckert, Kunle Odutola, Jeremy Bennett, Linus Tolke, and Michiel Van Der Wulp. ArgoUML User Manual: A Tutorial and Reference Description. Version 0.22. 2006.

[SPTJ] Gerson Sunye, Damien Pollet, Yves Le Traon, and Jean-Marc Jezequel. Refactoring UML Models. Proceedings of UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools, Volume 2185 of LNCS. Springer, pages 134-148. 2001.

[umb] Umbrello UML Modeller. `http://uml.sourceforge.net/index.php`. Date last accessed: 5th March, 2007.

[W3C07] W3C. XSL Transformations (XSLT) Version 2.0, 23th January, 2007.