A Template-Based Model Transformation Approach
Using A Simplified Hierarchical Metamodel

by

Hongming Liu

College of Computing and Digital Media

DePaul University

Advisor: Xiaoping Jia

Committee members:

Jane Huang

Jami Montgomery

Adam Steele

Konstantin Läufer
ACKNOWLEDGEMENTS

Undertaking this research has been a great learning experience. First and foremost, I’d like to thank my advisor, Xiaoping Jia, for his continuous support and guidance, for providing a very flexible working environment, and of course for his insight and dedication in this research project.

I am also greatly indebted to my committee members. I thank each of them for their extensive reviews and comments, and for making the time to be part of the committee in the first place: Jane Huang, Jami Montgomery, Adam Steele and Konstantin Läufer.

I appreciate the help of Lizhang Qin, Christopher Jones and Howard Dittmer. And thank them for their collaboration in the project and their support.

Last but not least, I want to thank my wife Pinqi Tan for her unconditional support and encouragement. I also thank my parents in China who lost the opportunity of higher education but have always instilled the loss as inspiration in me.
## Contents

1 Introduction 3

1.1 Problem Statement ................................................. 3
1.2 Approach ............................................................... 5
1.3 Summary of Contributions ........................................... 5
1.4 Outline of the Dissertation ......................................... 6

2 Background 8

2.1 Software Engineering and Software Development Process ............. 8
2.2 Model Driven Engineering ............................................ 9
   2.2.1 Model ........................................................... 10
   2.2.2 Metamodelling .................................................. 12
   2.2.3 Model Transformation .......................................... 13
2.3 Model Driven Architecture .......................................... 15
   2.3.1 UML ............................................................. 15
   2.3.2 XMI ............................................................. 17
   2.3.3 MOF ............................................................. 19

3 ZOOM Architecture and Notation 22

3.1 ZOOM Architecture .................................................. 22
3.2 Modeling with ZOOM Notation ....................................... 25
   3.2.1 Structural Models ............................................. 25
   3.2.2 Behavioral Models ............................................ 26
4 HRMT Model Transformation Approach

4.1 Source Model Representation ........................................ 30
   4.1.1 Hierachical Relational Metamodel .......................... 31
   4.1.2 HRM Definition .............................................. 32
   4.1.3 Path Expression Language in HRM ........................... 34

4.2 Transformation Template ........................................... 34

4.3 Transformation Algorithm ......................................... 37

4.4 Model Transformation Process .............................. 39
   4.4.1 Parsing the source model .................................... 40
   4.4.2 Traversing the object tree .................................. 41
   4.4.3 Matching node in object tree with rule ..................... 43
   4.4.4 Generating target text ...................................... 44
   4.4.5 Post processing ............................................ 47

5 The HRMT Tool Suite ........................................ 49

5.1 HRMT Architecture .............................................. 49

5.2 HRMT Main Components ........................................ 52

5.3 Use of HRMT Tool Suite ........................................ 54

6 Development of Transformation .................................. 55

6.1 Basic Classes .................................................. 55

6.2 Classes with Associations ...................................... 57
7 Case Study: A Hospital Information Management System

7.1 Introduction .............................................. 64
7.2 Requirement of HIMS .................................... 64
  7.2.1 Scope of HIMS ...................................... 64
  7.2.2 Functionality Requirements ......................... 65
  7.2.3 Non-Functional Requirements ....................... 66
  7.2.4 External Interface Requirements ................. 67
7.3 Modeling and Transformation of HIMS .................. 67
7.4 Features Illustrated by Case Study ..................... 68
  7.4.1 Multi-platform support ............................ 68
  7.4.2 Security requirement support ...................... 69
  7.4.3 Scalability requirement support ................... 70
  7.4.4 Performance requirement support .................. 72

8 Related Work ................................................. 74
  8.1 Direct-Manipulation Approach ......................... 74
  8.2 Transformation using XSLT ............................ 75
  8.3 Template-Based Approaches ......................... 77
  8.4 QVT(Queries/Views/Transformations) ............... 78

9 Evaluation of Transformation Tools ............................. 81
9.1 Transformation Tools Being Evaluated

9.1.1 Direct-Manipulation Approaches: Jamda

9.1.2 Transformation using XSLT: Stylus Studio

9.1.3 Template-Based Approaches: AndroMDA

9.1.4 QVT(Queries/Views/Transformations): ATL

9.2 Evaluation Result

9.2.1 Evaluation Result Reflecting Metamodel Complicity

9.2.2 Evaluation Result Reflecting Transformation Complicity

10 Conclusion

10.1 Contributions

10.2 Future Work

A Development of Transformation

A.1 Generation of Basic Classes

A.2 Generation of Classes with Associations

A.3 Generation of EJB Entities

A.4 Generation of Comprehensive EJB Applications

B ZOOM Model for Hospital Information Management System

C Screenshots of Hospital Information Management System
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>An Example of MDE Transformation</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>ZOOM Architecture</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>HRMT Model Transformation Overview</td>
<td>29</td>
</tr>
<tr>
<td>4</td>
<td>HRMT Model Transformation Process Overview</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>HRM Example of a Class Diagram</td>
<td>33</td>
</tr>
<tr>
<td>6</td>
<td>Graphic representation of HRM-Based Transformation Algorithm</td>
<td>38</td>
</tr>
<tr>
<td>7</td>
<td>Graphical Representation of PIM</td>
<td>41</td>
</tr>
<tr>
<td>8</td>
<td>Abstract Syntax of PIM</td>
<td>42</td>
</tr>
<tr>
<td>9</td>
<td>Model Transformation in details</td>
<td>43</td>
</tr>
<tr>
<td>10</td>
<td>Matching metamodel node and generating target text with rule</td>
<td>44</td>
</tr>
<tr>
<td>11</td>
<td>Expression Mapping in details</td>
<td>45</td>
</tr>
<tr>
<td>12</td>
<td>HRMT Architecture</td>
<td>50</td>
</tr>
<tr>
<td>13</td>
<td>ZOOM Graphical Representation of Hospital Information Management System</td>
<td>68</td>
</tr>
<tr>
<td>14</td>
<td>Model Transformation Process Comparison</td>
<td>95</td>
</tr>
<tr>
<td>15</td>
<td>ATL Approach</td>
<td>99</td>
</tr>
<tr>
<td>16</td>
<td>Evaluation Result of Source Model</td>
<td>102</td>
</tr>
<tr>
<td>17</td>
<td>Evaluation Result of Template</td>
<td>105</td>
</tr>
<tr>
<td>18</td>
<td>HIMS Main Page ASP.NET version</td>
<td>180</td>
</tr>
<tr>
<td>19</td>
<td>HIMS Main Page JSP version</td>
<td>181</td>
</tr>
<tr>
<td>20</td>
<td>HIMS View Medical Data ASP.NET version</td>
<td>182</td>
</tr>
<tr>
<td>21</td>
<td>HIMS View Medical Data JSP version</td>
<td>183</td>
</tr>
<tr>
<td>#</td>
<td>Table Title</td>
<td>Page</td>
</tr>
<tr>
<td>----</td>
<td>-------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>1</td>
<td>XMI Compatibility Test Result</td>
<td>19</td>
</tr>
<tr>
<td>2</td>
<td>HRM Components</td>
<td>33</td>
</tr>
<tr>
<td>3</td>
<td>Examples of PEL Expressions</td>
<td>34</td>
</tr>
<tr>
<td>4</td>
<td>Explanation of code fragment names</td>
<td>36</td>
</tr>
<tr>
<td>5</td>
<td>Example of Expression Mapping</td>
<td>46</td>
</tr>
<tr>
<td>6</td>
<td>Explanation of code fragment names</td>
<td>56</td>
</tr>
<tr>
<td>7</td>
<td>Explanation of code fragment names</td>
<td>56</td>
</tr>
<tr>
<td>8</td>
<td>Explanation of code fragment names</td>
<td>59</td>
</tr>
<tr>
<td>9</td>
<td>Explanation of code fragment names</td>
<td>60</td>
</tr>
<tr>
<td>10</td>
<td>Explanation of code fragment names</td>
<td>61</td>
</tr>
<tr>
<td>11</td>
<td>Explanation of code fragment names</td>
<td>62</td>
</tr>
<tr>
<td>12</td>
<td>Example of ZOOM Annotation</td>
<td>70</td>
</tr>
<tr>
<td>13</td>
<td>Explanation of Metric Factor</td>
<td>82</td>
</tr>
<tr>
<td>14</td>
<td>Evaluation Result of Source Model</td>
<td>101</td>
</tr>
<tr>
<td>15</td>
<td>Evaluation Result of Template</td>
<td>104</td>
</tr>
<tr>
<td>16</td>
<td>Evaluation Result of Transformation Tools</td>
<td>107</td>
</tr>
</tbody>
</table>
ABSTRACT

Model-Driven Engineering (MDE) is a model-centric software engineering approach which aims at improving the quality of and facilitating the development of software artifacts by focusing on the design models instead of code. Its vision is to build an abstract model of a system that we can then transform into more refined models and eventually transform into the system implementation. Although some variants of MDE, especially Model-Driven Architecture (MDA), are already quite advanced and serve as the conceptual foundation for commercial software products, there are many challenges to achieving true Model-Driven Engineering.

Two major challenges that researchers face when attempting to realize the MDE vision are 1) providing precise, analyzable, transformable, and executable models and 2) providing well-defined transformations that support rigorous model evolution, refinement, and code generation. Both of these challenges point to a mutual research topic: metamodeling. Metamodeling is the key technology that ensures precise, analyzable models, which is the basis for the transformation definition. Considering these challenges and their connection to metamodeling, we provide a solution that uses a simplified metamodel as the foundation for building a template-based model transformation framework. This simplified metamodel is called the Hierarchical Relational Metamodel (HRM). The Hierarchical Relational Metamodel is built upon Z-based Object-Oriented Modeling notation (ZOOM). HRM maintains both a tree structure and the relationships among model elements. The model elements and the tree structure are constructs of the ZOOM modeling language comparable to constructs of a programming language. To capture more complicated modeling language constructs like association, we adopt a mathematical collection to depict the relationships among different
constructs.

The design of the Hierarchical Relational Metamodel (HRM) provides us a simplified way to understand and make use of the abstract syntax and semantics of the source models. On top of the HRM, we design a template-based model transformation framework. The transformation framework provides a simplified way of getting information from the metamodel, makes the template definition more readable, and allows different sets of templates to be provided for model transformation between different target technical platforms.

A transformation tool, the Hierarchical Relational Metamodel Transformation tool (HRMT) that implements this design is developed. Its major components are discussed in this thesis. To demonstrate the ability of this tool, a case study is presented to show how HRMT can transform ZOOM model specifications to applications running on multi-platforms and to show HRMs support of non-functional requirements.

This thesis also provides a detailed study of some of the related work. A group of essential metrics is identified to evaluate different transformation tools. The result of the evaluation supports our argument that HRMT provides a simple, effective way to define model transformations and the benefits of this model transformation framework include readability of metamodel definitions and transformation definition.

In summary, we provide a model transformation approach that supports MDE using a simplified hierarchical metamodel. The benefits of this model transformation approach include 1) readability and rigorousness of metamodel definitions; 2) simplicity of transformation definitions; and 3) support of non-functional requirements.
1 Introduction

Model-Driven Engineering (MDE) tackles the problem of system development by promoting the use of models as the primary artifact to be constructed and maintained [1, 2]. MDE shifts software development from a code-centric activity to a model-centric activity. Accomplishing this shift entails developing support for modeling concepts at different levels of abstraction and then transforming abstract models to more concrete descriptions of software. In other words, MDE reduces complexity in software development through modularity and abstraction [3].

Because of MDE’s potential to dramatically change the way we develop applications, companies are already working to deliver supporting technologies [4, 5]. Although some variants of MDE, especially Model-Driven Architecture (MDA), are already quite advanced and serve as the conceptual foundation for commercial software products, there are many challenges to achieving true Model-Driven Engineering.

1.1 Problem Statement

The major challenges that researchers face when attempting to realize the MDE vision were discussed in [6]. According to many research projects that point out the inadequacy in MDE development [7–9], there are two main challenges that MDE infrastructure faces:

- Providing precise, analyzable, transformable and executable models [6].

- Providing well-defined transformations that support rigorous model evolution, refinement, and code generation [7].
The first challenge is currently addressed specifically yet inadequately by the Object Management Group’s (OMG) UML [10] standard specification. UML-2, with its current version UML 2.2, is the *de facto* standard object modeling notation for software engineering. It allows modelers to capture a wealth of information about software system components, their behaviors, and their interactions. However, currently UML-2 is insufficient for MDE owing to its deficiencies in several critical areas including incompleteness, semi-formalness and inconsistency [11, 12].

The second challenge, model transformation that support rigorous model evolution, refinement, and code generation is also an active research area [13, 14]. There are many research projects that provide fundamentals for model transformation. Model transformation is the process of converting one model into another model. Performing a model transformation requires a clear understanding of the syntax and semantics of both the source and target models. To take modeling to a higher level of abstraction, it is necessary to have a standard mechanism to define metamodels of modeling languages [15]. OMG addresses this issue in its MDE initiative Model Driven Architecture (MDA) [16, 17] by creating Model Object Facility (MOF) [18]. In response to the need for a standard approach to defining the functions that map between metamodels, the OMG issued the MOF 2.0 Query/View/Transformation (QVT) [19–21] Request for Proposals. Several replies were provided by a number of companies and research institutions that evolved over three years to produce a common proposal that was submitted and approved [21–23].
1.2 Approach

Both of the challenges mentioned above point to a mutual research topic: metamodeling. Metamodeling is the key technology that ensures precise, analyzable models, and it is the very element of metamodeling that is the basis for transformation definition. Considering these challenges and their connection to metamodeling, we provide a solution that uses a simplified metamodel as the foundation for building a template-based model transformation framework. This simplified metamodel is called the Hierarchical Relational Metamodel (HRM). The Hierarchical Relational Metamodel is built upon Z-based Object-Oriented Modeling notation (ZOOM). HRM maintains both a tree structure and the relationships among model elements. The model elements and the tree structure are constructs of the ZOOM modeling language comparable to constructs of a programming language. To capture more complicated modeling language constructs like association, we adopt a mathematical collection to depict the relationships among different constructs.

In summary, our objective is to apply such an overall research approach in realization of MDE focusing on model transformation. The notation and metamodel design lays the foundation of our tool development. And the development of the Hierarchical Relational Metamodel Transformation tool (HRMT) in turn demonstrates the validity and advantage of the design.

1.3 Summary of Contributions

This dissertation provides a solution that uses a simplified metamodel as the foundation to build a template based model transformation framework. This simplified metamodel is
called Hierarchical Relational Metamodel (HRM). The Hierarchical Relational Metamodel is built upon Z-based Object-Oriented Modeling notation (ZOOM). A template based model transformation framework using the Hierarchical Relational Metamodel (HRM) is introduced. This framework aims to provide a simple, effective, and practical way to define model transformations. The benefits of this transformation framework include 1) readability and rigorousness of metamodel definitions; 2) simplicity of transformation definition; and 3) support of non-functional requirements. A case study is provided to demonstrate the feasibility of the approach, and comparisons with related research work are provided to show the benefits of this framework.

The main contribution of our research project is the design and realization of such a model transformation framework. It fulfills the goal of transforming PIM to PSMs for different platforms. However, many research projects and commercial tools can claim the same thing. Our approach is distinguished from others by its simplicity of source modeling language, readability of transformation definition and simplicity of metamodel definitions. It also provides support of non-functional requirements.

1.4 Outline of the Dissertation

This dissertation is organized as follows: Chapter 2 provides detailed background information about model-driven engineering and model transformation. Chapter 3 presents the ZOOM architecture and notation. Then Chapter 4 covers the characteristics of our model transformation approach and elaborates on the model transformation process. Chapter 5 introduces the HRMT tool and its main components. In Chapter 6, several scenarios of transformation development are presented. Chapter 7 discusses a case study that demon-
strates our approach. Chapter 8 categorizes some of the related work to our research. Chapter 9 reviews the evaluation result of comparing our tool with others. Finally, Chapter 10 concludes the dissertation and summarizes the main contribution of our research work.
2 Background

2.1 Software Engineering and Software Development Process

Software Engineering, as defined in IEEE Standard 610.12, is “The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software”. Software engineering is an engineering discipline which is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use [24,25]. It is the branch of systems engineering concerned with the development of large and complex software intensive systems. It focuses on: the real-world goals for, services provided by, and constraints on such systems; the precise specification of system structure and behavior, and the implementation of these specifications; the activities required in order to develop an assurance that the specifications and real-world goals have been met; the evolution of such systems over time and across system families. It is also concerned with the processes, methods and tools for the development of software intensive systems in an economic and timely manner [26,27].

A traditional theme in software engineering discourse has been “why can’t we develop software like other engineers build bridges”. Software development is often compared with hardware development in terms of maturity and the progress in software development seems to be minimal [28]. However, the fact that it is feasible to build much more complex and larger systems today should have undermined this false sense of inferiority. As a result of academic and industrial research and experience in object-oriented design methods [29–31], formal methods etc, there have been significant improvements in the ways in which software
are built over the last fifty years [32–34].

Still software development is an area in which we are struggling with a number of problems. Writing software is labor intensive. With each new technology, much work needs to be done again and again. Systems are never built using only one technology and systems always need to communicate with other system. There is also the problem of continuously changing requirements. The most important problems with software development include productivity, portability, interoperability and maintenance problem [35–37].

2.2 Model Driven Engineering

The term Model Driven Engineering(MDE) was first proposed by Kent in [1]. MDE tackles the elusive problem of system development by promoting the usage of models as the primary artifact to be constructed and maintained. It aims for integration of software engineering phases such that the development of a system from the first abstract design to the finished product is a continuum with no gaps in between. Model driven engineering automates transformations between different models by using formal models of the software system and well-defined mappings (both horizontal and vertical) between different models [13,38].

Within the context of MDE, a model of the system is not just a drawing in some design document anymore but a set of formal concepts and elements [39]. A system may possess many different models which conceptualize different aspects of the system at different levels of abstraction. For example one model of the system may describe behavior of distinct components as finite-state automata whereas another model may describe collaboration between those components using a coordination model, such as shared tuple-spaces. In a traditional software engineering process, mapping between design and implementation is done by ex-
ploiting the expert knowledge of a programmer. In model driven engineering this expert
knowledge is not lost in millions of lines of code but instead it is explicitly visible in trans-
formation functions. Refinement steps from the first high-level design models all the way to
the implementation are given as formal mappings between models [6,40].

Model driven engineering can be exploited in all phases of the software design process [2,3].
Even very rough models can be used as an interface between customers and designers during
requirements analysis. During initial design and refinement phases, formal models can be
used to simulate, analyze or even automatically verify software designs. The ultimate goal of
MDE is the rapid development of robust software using reusable designs, or in other words,
the automatic generation of lower level models and complete implementation. Model driven
engineering tries also to achieve platform independent software solutions by using different
levels and views of models and modeling languages [41,42].

2.2.1 Model

Model driven engineering is a model-centric software engineering approach. In a MDE
system, a model is a consistent and complete set of formal elements describing a system that
is amenable to analysis [39]. For example, we could build a bank model, ignoring security
and user interface aspects, or we could model a combination of these domains. We choose
which subject matters to include and which to ignore, although we might then need to weave
several models together. When a model’s subject matter has a high degree of abstraction,
the model is closer to the eventual user’s language that is, a smaller gap exists between a
non computer expert and the model. A model is expressed in a language that exists at some
(language) abstraction level. A model written in the C modeling language will ignore (or
abstract away) the realization of function calls and expressions, leaving CPU-oriented issues such as register allocation to the compiler or to a virtual machine interpreter that adds such realizations at runtime. Similarly, a model expressed in the Unified Modeling Language will ignore the realization of associations, leaving those decisions to the model compiler or human designer [39].

Model driven engineering offers the potential for automatic transformation of high-level, abstract application subject matter models into running systems. Researchers have pointed that modeling technology has matured to the point where it can offer significant leverage in all aspects of software development [43, 44]. In an increasing number of application areas, you can generate much of the application code directly from models. Automation doesn’t remove the requirement for creativity. Rather, it formalizes existing solutions and raises the level at which we can apply creativity, thus giving the developer more leverage. To make automation a reality, models must have a defined meaning, or in other words, be written in a well defined language [45, 46]. A language consists of syntax and semantics. The syntax can be human or machine centric. Semantics define what the syntax means by linking the syntax to a semantic domain [47]. Model driven engineering automates the transformation of models from one form to another. We express each model, both source and target, in some language. The target models language, for example, might define a meaning for remote access of objects, even though the source models language does not. We must clearly define the two languages. Because modeling is an appropriate formalism to formalize knowledge, we can define a modeling language’s syntax and semantics by building a model of the modeling language - a so-called metamodel. We will cover more about metamodel in the following section.
2.2.2 Metamodeling

As we stated above, a model in the context of MDE needs to be written in a well-defined language. A well-defined language was defined as a language which is suitable for automated interpretation by a computer. Metamodeling can help define a well-defined language [48,49].

Languages are often defined using a grammar in Backus Naur Form (BNF), which describes what series of tokens is a correct expression in a language [50]. This method is suitable and heavily used for text-based languages, like programming languages. A BNF grammar could be used to define modeling language. It does fulfill the requirement that it is suitable for automated interpretation. Because modeling languages do not have to be text based, and often aren’t (they can, for example, have a graphical syntax, like UML). A different mechanism for defining languages in the MDE context will be needed. This mechanism is called metamodeling.

Metamodel is defined in [39] as “a specification model for a class of systems under study where each system under study in the class is itself a valid model expressed in a certain modeling language”. In other words, a metamodel describes the abstract syntax of a modeling language as a class model. Each class in a metamodel describes a model element, i.e. a concept or abstraction in our modeling language. Every kind of element that a modeler can use in his or her model is defined by the metamodel of the language the modeler uses. Because a metamodel is also a model, a metamodel itself must be written in a well-defined language. This language is called a metalanguage. For example, BNF is a metalanguage. A metalanguage plays a different role than a modeling language in the MDE framework, because it is a specialized language for describing modeling languages. Because a metalanguage is a language itself, it can be defined by a metamodel written in another metalanguage. In theory
there could be an infinite number of layers of model-language-metalanguage relationship [51].

2.2.3 Model Transformation

Model driven engineering moves development focus from third generation programming language code to models [13,52]. The objective is to increase productivity and reduce time-to-market by enabling development and using concepts closer to the problem domain at hand, rather than those offered by programming languages. The key challenge here is to transform these higher-level models to platform-specific models that tools can use to generate code.

Figure 1 shows the overall picture of MDE using an example. On the top is a Platform Independent Model(PIM), which is transformed into three different Platform Specific Model(PSM) in three different platforms: Relational Database, EJB and .NET [53–55]. In MDE the focus of a developer shifts to the development of a PIM. The PSMs that are needed are generated by transformation from PIM into multiple PSMs for different platforms. Everything specified at the PIM level is therefore completely portable. And the fact that PSM can be transformed to code automatically greatly improves productivity.
For the model driven engineering vision to become a reality, tools must support model transformation [55–57]. Development tools should let users not only apply predefined model transformations but also define their own. Beyond automating transformation execution, tools could suggest which model transformations a user might appropriately apply in a given context [58]. Performing a model transformation requires a clear understanding of the abstract syntax and semantics of both the source and target. As introduced in the last section, metamodeling is a common technique for defining the abstract syntax of models and the interrelationships between model elements. For visual modeling languages, there are a number of advantages in basing a tool’s implementation on the languages metamodel [59]. Such tools offer users three different architectural approaches for defining transformations:

- Direct model manipulation: tools access an internal model representation and have the ability to manipulate the representation using a set of procedural APIs. Jamda is an example of this approach [60].

- Intermediate representation: this approach exports the model in a standard form, typically XML, so an external tool such as XSLT can transform it [61,62].

- Transformation language support: many tools use this approach and provide a template-based or rule-based transformation language that provides a set of constructs for explicitly expressing, composing, and applying transformations [63–65]. This approach is also what our transformation framework adopts.
2.3 Model Driven Architecture

Model Driven Architecture is an initiative by the Object Management Group to address the issue of model driven engineering [66]. MDA is a new way of writing specifications and developing applications, based on a platform independent model (PIM). A complete MDA specification consists of a definitive platform independent base UML model, plus one or more platform specific models (PSM) and interface definition sets, each describing how the base model is implemented on a different middleware platform [67].

MDA development focuses first on the functionality and behavior of a distributed application or system, undistorted by idiosyncrasies of the technology or technologies in which it will be implemented. MDA separates implementation details from business functions. Thus, it is not necessary to repeat the process of modeling an application or systems functionality and behavior each time a new technology comes along. While other architectures are generally tied to a particular technology, MDA is trying to model functionality and behavior once and only once. Mapping from a PIM through a PSM to the supported MDA platforms will be implemented by tools, easing the task of supporting new or different technologies [68,69].

The approach I am proposing is parallel with MDA in many ways. However, the essential difference is the underlying notation and metamodel design. In the following two sub sections, we will look at MDA’s underlying notation UML and it’s metamodel MOF.

2.3.1 UML

In the last few years OMG’s Unified Modeling Language (UML) has emerged as the standard modeling notation of the software industry [29,70]. UML is used in the development process of various object-oriented and especially component-based systems. It provides the
software developer with the facility to specify, visualize and document large system models, including the static application structure as well as dynamic behavior of the system. Additionally it allows describing the instantiation of objects and their distribution. Right now the prominence of UML is still augmented by the OMG’s MDA standardization process, where UML is utilized as a general purpose language for various domains on different abstraction levels [71].

UML is a graphical notation that provides the software developer with twelve diagram types, basically classified into three categories:

- Structural Diagrams represent static application structure (Class Diagram, Object Diagram, Component Diagram, and Deployment Diagram)
- Behavior Diagrams describe aspects of dynamic behavior (Use Case Diagram, Sequence Diagram, Activity Diagram, Collaboration Diagram, and State-chart Diagram)
- Model Managing Diagrams illustrate how to organize and manage your application modules (Packages, Subsystems, and Models)

UML as a core standard of MDA enables the development of platform independent models (PIM) as well as platform specific models (PSM). Naturally a software developer starts to describe the business semantics of his application by building a PIM, which captures the structure and behavior of the application but no technical aspects. Later on the PIM is transformed by OMG-standardized mappings into a PSM for the desired platform. Normally this transformation is accomplished by an MDA-enabled development tool. In a final step the tool generates the source code from the information covered by the PSM.
In his keynote speech in Codegen conference 2008 [72], Bran Selic points out that the domains are getting increasingly more specialized requiring increasing language specialization. While UML 2 is designed as a family of modeling languages, it uses semantic variation points to address this issue.

Fundamentally the flexibility and compensability of MDA is based on the generic usability and methodology- independence of UML. Thereby programmers are able to exchange and combine PIMs and PSMs developed with different tools on variable platforms. One can even interchange UML models and above all their graphical information between different tools by using XMI (XML Metadata Interchange, another OMG standard in progress) [73–75].

2.3.2 XMI

The Object Management Group (OMG), the same standard organization that maintains the UML standards, proposes the use of XMI and XMI-DI [74] to enable model interchange. One of the strong points of XMI is that it is based on XML [73]. XML has been successfully used to support many document and model representation standards. XML is well-documented, machine-independent and there exist plenty of tools supporting it. Thus, XMI documents should be portable and easy to parse.

It is also important to remark what XMI is not. XMI cannot be used to define the structure of a modeling language. This is the role of the Meta Object Facility (MOF) [76] standard, which will be covered in the following subsection. Actually, XMI can be used to serialize any MOF-based modeling language including, but not restricted to UML and MOF itself. The fact that XMI is independent of a modeling language is at the same time one of its strong points and major weaknesses. First, two tools that use two different versions of the UML
standard, for example UML 1.4 and UML 2.0 will not be able to exchange models even if
they use XMI. Also, a UML model serialized as a XMI document will not contain any extra
information that is not present in the UML model as such.

A study in [77] pointed out that one of the main impediments to use XMI in practice is
the large number of different versions of the XMI standards and UML metamodels. At the
moment of writing this text, there are five versions of XMI (1.0, 1.1, 1.2, 2.0 and 2.1) and
eight versions of UML (1.0 to 1.5 and 2.2). This means that a document containing a UML
model can actually be serialized in 40 different combinations of XMI and UML versions.
The diversity of different XMI implementations has already been studied in [78]. There,
Jiang devised a method to explore differences between XMI formats using DTDs. The fast
evolution of XMI and UML gives little room for implementors to try it out in practice.

In an article from the MODELS conference, Lundell et al. [79] tested XMI with 14 UML
tools. Obviously, the older tools cannot load from the newer tools, but can the newer tools
read models from the older tools? That’s the important question after all, if you want to
use XMI as insurance against your tool being discontinued: will this year’s tools load last
year’s models? The result is shown in Table 1, see the article for full details.

In this study, eight older tools are used for testing: ArgUML, Fujaba, Artisan, Poseidon,
Rhapsody, Rose 1.0, Rose 1.1 and Visio. They are listed in the first column of Table 1. XMI
files are exported from these tools. Five UML modelling tools were identified which support
XMI version 2.0 (or later): Borland Together Architect 2006 for Eclipse, EclipseUML Free
Edition, IBM Rational Software Architect 6.0, IBM Rational Software Architect 6.0 and
Altova UModel 2006. They are listed in the first row of Table 1. They will import XMI
files and the failed or successful results are recorded in the table. For example, the XMI
file exported by ArgoUML failed to be imported by Borland while the XMI file exported by Fujaba was successfully imported by Borland.

The results show unsuccessful interchange for the majority of tool combinations. There are only two successful cases out of all 40 combinations. This result very clearly exposes the problem of XMI compatibility.

### 2.3.3 MOF

UML defines models, which have their own information, and they have meta-information pertaining to them. Although UML defines its own metamodel for expressing its meta-information, the OMG also has the Meta-Object Facility (MOF) as a meta-information management standard, which evolved from needs of expressing metadata for CORBA in platform independent ways [80]. Nowadays the MOF defines a generic framework for describing and representing meta-information [81].

The OMG MOF is designed for use in a wide variety of scenarios that have a significant metadata component; e.g. type management, software development, information management and data warehousing. MOF has contributed considerably to some of the core principles
of the MDA. Building on the modeling foundation established by UML, MOF introduced
the concept of formal metamodels and Platform Independent Models (PIM) of metadata
(examples include several standard such UML, CWM, EDOC, EAI etc.) as well as mapp-
ings from PIMs to Platform Specific Models (PSM) such MOF-to-IDL mapping in the MOF
specification or MOF-to-XML DTD mapping in the XMI specification [82–84].

The core of the MOF approach to meta-data management is openness. The goal is to
provide a framework that supports any kind of meta-data, and that allows new kinds to
be added as required. In order to achieve this, the MOF specification adopts a four-layer
metamodeling architecture:

- M0 What is to be modeled
- M1 Models (for example, a UML model)
- M2 Metamodels (for example, an abstract syntax model in the UML specification)
- M3 The meta-metamodel (MOF)

Layer M3 is a specification of the modeling language used to express metamodels in M2.
In OMG, this is always the MOF metamodeling language. This metamodeling language is
reflexively specified, so no higher layers are needed. When a modeling language’s metamodel
uses that same modeling language, the statements in the metamodel are expressed in the
same language the metamodel is describing. We call this a reflexive metamodel. This meta-
model circularity has raised some concern by researchers [85, 86]. To break the circularity,
definition ultimately must be provided in terms of more basic concepts, for example via a
formal mathematics semantics or an explicit set of axioms and deduction rules.
OMG also provides a generic concrete syntax called Human-Usable Textual Notation (HUTN) [87] for MOF-based metamodels. HUTN defines a generic concrete syntax, which aims to conform to human-usability criteria. However, as [75] pointed out that there were no current reference implementation of HUTN: the Distributed Systems Technology Centres TokTok project (an implementation of the HUTN specification) seems to be inactive (the source code can no longer be found), whilst work on implementing the HUTN specification by Muller and Hassenforder [88] has been abandoned in favor of Sintaks [89], which operates upon domain-specific concrete syntax. Instead, metamodel developers use generation tools to derive specific syntax parsers and editors. Among these approaches, Eclipse technologies such as EMF [90] (and GMF [63]) can be used to generate tree-based (and graphical) model editors. Baar [91] introduces a formalization of visual concrete syntax, and highlights its usefulness in analysis of correctness. Text-to-model transformation tools such as TCS [92] and TEF [93] can be used to define a custom concrete syntax for a metamodel and to generate a corresponding parser.
3 ZOOM Architecture and Notation

Since our Hierarchical Relational Metamodel is built upon Z-based Object-Oriented Modeling notation (ZOOM), it is necessary to introduce this notation and overall ZOOM architecture. The following sections present the architecture and main components of ZOOM. It is also important for us to point out that although our HRMT approach is developed based upon ZOOM, the idea of creating a simplified metamodel to facilitate model transformation is independent of ZOOM. In other words, our HRMT can be implemented upon another notation as long as we can build a hierarchical relational metamodel on top it. However, since ZOOM is in fact the foundation of HRMT tool, we will limit the discussion of underlying notation to ZOOM only.

3.1 ZOOM Architecture

While UML-2 is widely used as a visual modeling language to support MDE, it has several weaknesses. UML-2 is not specifically designed for MDE, so its models are generally informative without providing definitive views. Also, while UML-2 provides multiple visual views to present similar aspects of a software design, it lacks an inherent mechanism to enforce consistency between these views. Additionally, UML-2 lacks user and system interface design notations. To overcome these obstacles, and to enhance the UML-2 models and metamodel to include support for formal syntax and semantics and to provide a new UI model, a new formal notation called ZOOM is developed in an earlier research project [94]. ZOOM stands for Z-based Object-Oriented Modeling. It is based on the formal specification notation Z [95–99], which is in turn based upon set theory and mathematical logic. ZOOM
provides a human-readable syntax to specify the mathematical model in Z. A complete description of how ZOOM supports Z notation can be found in [94]. Although widely used to specify software systems, one deficiency of Z is that its specification is limited to mathematical logic and does not provide useful mechanisms to support OO modeling such as classes or inheritance. ZOOM extends Z to support these object-oriented concepts. Another deficiency of Z (and the other Z-Based OO extensions) is the lack of visual notations for its constructs and an absence of notations for specifying user interfaces. ZOOM provides visual representations of models that are consistent with UML-2, and extends those notations to support UI and formal action design.

Figure 2 shows the overall architecture of the ZOOM models for a software system. The functional requirements derive the structural, behavioral and UI models. ZOOM provides a pre-defined event model, which is processed by an event-driven framework, to bind the structural, behavioral, and UI models together [100]. The integrated ZOOM model will be processed by the Knowledge-based Model Compilation Tools resulting in different implementations of the software system based on the specific platform and knowledge base.

We partition the software design into three components: structural models, behavioral models and UI models. The separation of a system into these three parts is an application of the well-known paradigm in software engineering - Separation of Concerns, which formally separates the system based on special purpose concerns [101]. This separation allows each aspect of the system to be specified separately, making each aspect easier to write, understand, and change with less impact on the other aspects. For example, under this separation, modelers can modify the user interface based on device profiles and user preferences without changing the structural or behavioral models. The other advantage of this separation is
that we can use different formal specification languages to describe different aspects of the system. Using a specific language, which is developed and aimed at a specific need, makes the modeling process more accurate and formal.

Each ZOOM model has dual representations including a textual specification and a visual view. The visual representations are consistent with common UML-2 diagrams, such as class, use case and state machine diagrams, but also include semantics and extensions. This enables the use of popular tools to design and maintain ZOOM models. Modelers will appreciate the ability to use available tools to construct their models and to add formal specifications to those models. Unlike the models in UML-2, the ZOOM models are executable with its execution semantics. All three components in ZOOM can be independently animated, and the whole software system can be visually animated with the event-driven framework.

Figure 2: ZOOM Architecture
3.2 Modeling with ZOOM Notation

ZOOM modeling notations consist of three components: structural models, behavioral models and UI models, each of which is discussed in detail in the following sections.

3.2.1 Structural Models

The structural model defines the functional part of the system using object-oriented concepts. These models provide not only the class relations and hierarchies, but also a precise specification of the functionality of each entity. Structural models are specified by using ZOOM-M language. Graphically, the visual modeling notations are consistent with the UML-2 class diagram. Textually, the ZOOM-M syntax is consistent with many Java language constructs. The main characteristics of ZOOM-M include formalism, the use of object-oriented constructs, and side-effect free operations. ZOOM-M is strongly typed with a semantically rich type system that supports inheritance and generic types [94]. ZOOM-M includes a library that contains a collection of pre-defined structural models, including Set, List, OrderedSet, Bag, Relation, Map, Pair, Tuple, etc. Through this library and the quantified iterator expression grammar, ZOOM-M is able to specify everything that Z can specify. Syntactically, ZOOM-M provides many language constructs like attributes and operations, which are similar to Java and C++ expressions. This makes the models easily understandable by industry developers. Additionally, ZOOM-M provides a concrete and precise syntax, which extends OCL. ZOOM-M is as expressive as Z and OCL but with a friendlier syntax and a much richer set of libraries [102, 103].

Each ZOOM structural model is called a struct. Structs are the basic building blocks in ZOOM-M. They are class-like entities, providing attributes and operations. Structs are
defined with formal specifications including class invariants and operation pre- and post-
conditions. These constraints are a key element of Betrand Meyers principle of design by
contract [104], which provides elements of the ZOOM foundation. The use of such constraints
can improve the quality of software documentation as well as its precision [105].

3.2.2 Behavioral Models

The behavioral model is the central communication mechanism that links the structural
models with the UI models. It uses an extended and formalized state machine diagram to
specify the dynamic aspects of a system.

In UML-2, there are five kinds of diagrams used to specify the different behavioral char-
acteristics of systems. Among these, State Machine diagrams and Activity diagrams are
widely used. Although these two views can be used to illustrate behaviors, they have their
own weakness when considering a fully automatic MDE approach. A state machine diagram
represents the lifecycle of an object, therefore it is good for specifying the control object,
but it lacks the ability to show the collaboration of multiple objects. An activity diagram
defines the behavioral logic of one activity based on data flow, but more closely resembles
low-level programming constructs rather than a high-level abstraction of the design.

Compare to UML-2 state diagram, ZOOM behavioral model has several improvements.
It introduces the variable declaration in state machine; an optional “extends” semantics
is introduced for state machine declaration; actions are defined using formal pre / post-
conditions, ZOOM-SM or activity diagram; It also provides a ”post” action, which is used
to fire new event to behavioral models; the parameter passing and naming mechanisms are
introduced in activity diagram; a formal event description language is defined. Formalized
state machine plus those improvements defines a formal and powerful behavioral modeling notation.

3.2.3 User-Interface Models

Unlike UML-2 and OMGs MDA notations, ZOOM separates UI models from other aspects [106]. This comes from the intrinsic characteristics of the UI aspect being separated from the structural and behavioral aspects. This separation is useful even when we consider external systems. In ZOOM, we consider all external systems as a special type of user. Those external systems, for example, a database system, communicate with the designed software through the same mechanism as the human users do. The UI typically has a tight coupling to a specific platform while other aspects are typically platform-independent [107]. ZOOM uses a specific visual UI modeling notation to achieve this separation of concerns rather than mixing the UI design as part of programming. One advantage of this separation is that it is easy to change a UI view by only changing the UI models, thus leaving other aspects unchanged. This is especially important while developing multi-platform software systems. Another advantage is that it is convenient for a modeler to model the software system from the bottom-up. While a UI model is typically composed of several nested sub-UI models, each of them can find the structural and behavior counterparts as a subset of the whole structural and behavioral models. A second advantage of this separation of concerns is that we overcome the difficulties of specifying the UI together with the structural model. In its present form UML-2 fails to enforce this separation of concerns, trusting instead in the capabilities of the modelers or the modeling tools.
4 HRMT Model Transformation Approach

The primary objective of our approach is to assist software development, from platform independent models to platform dependent models and code, with a framework that provides a simple way to define transformations, mappings, and refinements. We accomplish this by using ZOOM as a UML extension to define a platform independent model and a transformation mechanism supporting model-to-model transformation.

The use of metamodeling techniques provides a way for transformation to understand and make use of the abstract syntax and semantics of both the source and target models. This allows the transformation to happen at the metamodel level. Figure 3 shows the overview of our model transformation approach. It shows that the Transformation Template is defined in M2, the metamodel level, while the Transformation Execution is implemented in M1, the model level. Also shown in the diagram, the ZOOM Metamodel is defined by a Hierarchical Relational Metamodel(HRM). Metamodeling is the key in this structure. First, it provides a mechanism to define modeling languages such that they are unambiguously understood. A transformation tool can then read, write and understand the models. Second, the transformation rules that constitute a transformation definition describe how a model in a source language can be transformed into a model in a target language. With metamodeling, these rules use the metamodels of both the source and target languages to define the transformation.

Our approach to model transformation is different from most of the existing MDE approaches that are based on MOF. We propose a simpler hierarchical meta modeling architecture than MOF. The key element in our model transformation is how transformations
can be specified at the metamodel level. The approach to specifying model transformations involves specializing the Hierarchical Relational Metamodel (HRM) we proposed to characterize source or target models. We will discuss the use of the metamodel in transformation in the following sections.

Figure 4 shows the basic structure of our model transformation framework. A transformation engine takes the HRM defined source model as input, and uses a template comprised of a set of transformation rules to produce an output model in a format specified by the templates. In other words, the output from the transformation engine is a transformation of the input model. We regard a model as a set of model elements that are in correspondence with a metamodel element via the instantiation relationship. Metamodel based transformations use only the elements of the metamodels, thus the transformation description is expressed in terms of the two metamodels. The rule set in the Figure 4 is the transformation template which is an extensible component. Different set of templates can be used in different transformation tasks for various target platforms. It’s in this sense that we also call the
template “cartridge” to reflect the exchangeability of templates. It’s the core component of the transformation framework. We will discuss the characteristics of our approach in the following subsections.

4.1 Source Model Representation

We use ZOOM notation to represent the Platform Independent Model(PIM). ZOOM notation has a textual syntax defined by BNF, which gives us a simplified way to define and use the ZOOM metamodel. Since ZOOM provides the textual syntax in a form that most programming languages have, we are able to build an internal representation of ZOOM models in a structure similar to an Abstract Syntax Tree(AST), except that the nodes in the tree will be constructs of the modeling language instead of constructs of the programming language. However, to capture more complicated modeling language constructs like association, we adopt mathematical collections to depict the relationships of different constructs. Considering its tree structure and such relationships, we name our metamodel Hierarchical Relational Metamodel(HRM).
The use of HRM provides a way for transformation to understand and make use of the abstract syntax and semantics of both the source and target models. Based on HRM, we design our template based model transformation to get the information necessary to generate the target model or code from HRM-compliant models inside a model repository. A set of interchangeable templates can be provided for model transformation between different target technical platforms.

Metamodeling is a critical part of our transformation approach. It provides a mechanism to unambiguously define modeling languages - ZOOM in our case. It is the prerequisite for a model transformation tool to access and make use of the models. We will now look into the design of our Hierarchical Relational Metamodel (HRM).

4.1.1 Hierarchical Relational Metamodel

The fact that ZOOM notation has a textual syntax defined by BNF gives us a simplified way to define and use the ZOOM model’s metamodel. From an implementation point of view, metamodel defines the internal representation of models. In programming languages, this internal representation often takes the form of Abstract Syntax Tree (AST) that can be processed by an interpreter or compiler [108, 109]. Since ZOOM provides the textual syntax in a form that most programming languages have, we are able to build an internal representation of ZOOM models in a structure similar to an Abstract Syntax Tree. The only difference is the nodes in the tree are constructs of the modeling language instead of constructs of programming language. To capture more complicated modeling language constructs like association, we also adapt mathematics collection to depict the relationships of different constructs. It is considering its tree structure and such relationships that we
name this metamodel Hierarchical Relational Metamodel (HRM).

This tree structure of the metamodel is the fundamental difference between HRM and MOF. MOF uses a graphic structure that makes it harder to represent relationships between elements of the metamodel. Since model transformation uses the representation of metamodel elements to gather information about the source model, a more complicated representation will make the model transformation task more complicated as well.

4.1.2 HRM Definition

We provide the following definition of HRM:

**Definition 4.1.** Hierarchical Relational Metamodel is a 3-tuple: $HRM = (N, C, R)$, where

- $N$ is a set of nodes: $N = \{n_1, n_2, \ldots, n_j\}$
- $C$ is a relation on $N \times N$, which forms a tree structure that has one root and no unconnected nodes. Each node may have zero or more children. In other words, a node is either a leaf (i.e. with no children) or can be decomposed as one or more children and each child forms a subtree. Essentially HRM use composition relationships as the central relationship to capture the hierarchical structure of the metamodel.
- $R = \{r_1, r_2, \ldots, r_k\}$ is a set of relations between nodes, where $r_i$ is a relation on $N \times N$.

Figure 5 shows a simple class diagram that has four classes: Student, Graduate, Undergraduate and Course. The corresponding HRM diagram is also show in Figure 5 in the middle. This metamodel can be represented as $(N, C, R)$ according to Definition 4.1. More specifically, we can elaborate the contents of its three components in Table 2. The components $r_1, r_2, r_3$ and $r_4$ are relations between classes $n_1, n_2, n_3, n_4$ and relationship enroll,
The tree structure of HRM is the fundamental difference between HRM and MOF. MOF uses a graphic structure that makes it harder to represent relationship between elements of the metamodel. Reflected in XMI, the textual representation of MOF based model, there are no composition relationships separately maintained within a model element. This lack of a simple structure requires MOF based model transformation tools to use their own mechanism.
Table 3: Examples of PEL Expressions

<table>
<thead>
<tr>
<th>PEL Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>n.this</td>
<td>node n3: Graduate</td>
</tr>
<tr>
<td>n.parent</td>
<td>ClassDiagram</td>
</tr>
<tr>
<td>n.children</td>
<td>nodes n3.advisor and n3.thesis</td>
</tr>
<tr>
<td>n.elementName</td>
<td>value of the attribute elementName of n3: Graduate</td>
</tr>
<tr>
<td>n.children[0].elementName</td>
<td>value of the attribute elementName of n3.advisor: advisor</td>
</tr>
<tr>
<td>n.children[1].elementName</td>
<td>value of the attribute elementName of n3.thesis: thesis</td>
</tr>
</tbody>
</table>

to build simplified metamodel representation.

4.1.3 Path Expression Language in HRM

The tree structure in the metamodel makes it convenient to navigate inside a specific model. We use a sub-set of the ZOOM notation, Path Expression Language (PEL) to accomplish the navigation. There are two key features of PEL. One feature is to address parts of HRM. It is in a format similar to XPath [110,111], which is a syntax for defining parts of an XML document. Each expression is under a particular context node. The other feature of PEL is to support ZOOM syntax variable and expression evaluation. Since we have developed a fully functioning ZOOM Evaluator, the transformation process can call the ZOOM Evaluator to get the results of PEL.

For example, using the HRM diagram shown in Figure 5, if \( n = n3 \) is the current context node in a HRM tree. PEL supports the expressions shown in Table 3.

4.2 Transformation Template

The rule set shown in Figure 4 is a collection of transformation rules. This rule set is functioning as a container of transformation templates. Here we provide the definition of
transformation rule as followings:

**Definition 4.2**: A transformation rule set \( R = \{ r_1, r_2, \ldots, r_n \} \) is a set of model transformation rules.

**Definition 4.3**: A transformation rule \( r = P \rightarrow (T_{pre}, T_{post}) \) where \( P \) defines the pattern to select the element of the source model and the template pair \( (T_{pre}, T_{post}) \) defines the mapping to target model. Respectively \( T_{pre} \) defines the mapping to target model before traversing children of the selected element, and \( T_{post} \) defines the mapping to target model after traversing children of the selected element. The rationale of this design is closely related to the transformation algorithm that we will talk about in the next sub section.

In our framework, the development of transformation is in a large part the process of constructing transformation rules. The rule set in the Figure 4 is an extensible component. Different sets of templates can be used in different transformation tasks for various target platforms. That’s why we also call the template a “cartridge” to reflect the exchangeability of templates. The template is the core component of the transformation framework. We will show how template or rules are developed in Section 7.

Model transformation is the notion that we can transform a source model of a system into a real system in one or more target platforms. The source model contains information about the system entities you want to generate but it doesn’t have the knowledge of syntax requirement of model or code in a target platform. Model transformation template is designed to fill in this missing knowledge and the template represents the syntax of the target modeling or programming language. So the development of transformation is actually the process of constructing the transformation template.

Definitions 4.2 and 4.3 give the definition of transformation template and transformation
rules. To further understand the structure of a rule, let’s look at a simple example of a rule for transforming a “struct” in ZOOM model to a “class” in Java.

\[
r = (P \rightarrow (T_{\text{pre}}, T_{\text{post}}))
\]

\[
P = \text{struct}
\]

\[
T_{\text{pre}} = (\text{File, Import, CON}_{\alpha}, \text{FIELD}_{\alpha}, \text{GetSet}_{\alpha})
\]

\[
T_{\text{post}} = ()
\]

In the above specification, \( P = \text{struct} \) specifies the element of the source model the rule will be applied to. \( T_{\text{pre}} \) is described using a group of code fragment names which function as code fragment placeholders. A code fragment placeholder is an anchor to reserve a place for more detailed code fragments. The using of such concise and friendly names to depict fragments of code avoids the details of concrete syntax and facilitates the understanding of the construction of rules. The same mechanism will be used in section 7 when more transformation development is discussed. Table 4 shows the meaning of each of these names.

<table>
<thead>
<tr>
<th>Name</th>
<th>Code Fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>java file creation</td>
</tr>
<tr>
<td>Import</td>
<td>import list</td>
</tr>
<tr>
<td>CON(\alpha)</td>
<td>placeholder for class constructors</td>
</tr>
<tr>
<td>FIELD(\alpha)</td>
<td>placeholder for class member of fields</td>
</tr>
<tr>
<td>GetSet(\alpha)</td>
<td>placeholder for get and set methods</td>
</tr>
</tbody>
</table>

Using a template is one of the advantages of our transformation framework. The flexibility of using different templates for different target systems makes ours an extensible architecture. The interchangeability of different templates is parallel with cartridges in a printer system. In our transformation framework, there are three ways to obtain a template or cartridge: you can write it from scratch; you can use a pre-defined template “as is”; or you can extend a predefined template in a limited way.
4.3 Transformation Algorithm

Metamodel based transformation uses the elements of metamodel. Our adopting of Hierarchical Relational Metamodel (HRM) allows us to build an internal representation of ZOOM models in a structure similar to an Abstract Syntax Tree (AST). Once metamodel is generated as an AST like structure, it is accessible by the transformation process through traversing the tree.

We use an algorithm of “pre-order” to traverse the tree which means each node is visited before its children are visited and the root is visited first. The algorithm is exemplified by the pseudo code shown in Listing 1. This algorithm shows that for each node, transformation involves the following in order:

- finding the matching rule for this node (line 2)
- getting the output text by applying pre part of the rule (line 3, 4)
- traversing all the children nodes of this node and performing a transformation on each of them (line 5-7)
- getting the output text by applying post part of the rule (line 8, 9)

As we can see in Definition 4.3, a transformation rule has two mapping parts, $T_{pre}$ and $T_{post}$. They are represented as $rule.pre$ and $rule.post$ in the pseudo code. Shown in Listing 1: $rule.pre$ is the mapping before traversing children of the selected element, while $rule.post$ is the mapping after traversing children of the selected element. As shown in line 5-7 in Listing 1, each node in the metamodel will be visited once and all its children node will get
transformnode(node, RuleSet, Output_Model)

rule ← findMatchingrule(node, RuleSet)

targettext ← instantiate(rule.pre, node)

OutPut_Model.append(targettext)

foreach c is a child of node
    transformnode(c, RuleSet, Output_Model)

endforeach

targettext ← instantiate(rule.post, node)

OutPut_Model.append(targettext)

Listing 1: HRM-Based Transformation Algorithm

Figure 6: Graphic representation of HRM-Based Transformation Algorithm

visited. To trigger the transformation algorithm, the root node of the source metamodel needs to be passed, in the form of transformnode(root, RuleSet, Output_Model).

Figure 6 shows a graphic representation of this algorithm. 6 nodes are shown in the graph. The number in the circle shows the visiting order. Each node is visited twice. In each visit, an instantiation process will use information provided by both the node and rule to generate target text.
4.4 Model Transformation Process

To illustrate how our approach works, we will discuss the model transformation process in more detail by using a real example. Although the example is not completely detailed out for all parts of the system, complete code and rules are giving in appendix. The goal here is to show how the transformation process is carried out step by step, so we will not elaborate most of the process specific implementation details.

Community Sports League (CSL) is an organization that facilitates sport games between regional communities. There are 10 teams in CSL right now. Each team has 12-15 regular players. While CSL is gaining recognition and popularity, they are thinking of creating a system to manage team and player information and put some of the information online as well. Basically, they want to centralize the management of team and player information and they want people to be able to browse through the teams and look up a certain player. With a couple of mouse clicks, people can see their favorite local sport star’s weight, height and recent game stats.

To start the MDE process we need to build a platform-independent model that comprises the essence of CSL system. This is the only model that the developer will create completely “by hand.” The other models are mostly generated. The PIM for Community Sports League is depicted in Listing 2. It will be used as input in the following code generation example as Roster.zoom.

The PIM is also demonstrated in Figure 7 with its graphical representation. Figure 8 shows the internal representation of abstract syntax of the PIM. As you can see, the components, properties and relations existing in the model Roster are defined in the metamodel in a tree
Listing 2: Roster.zoom

structure. The texture representation can be given in XML as the right side of Figure 8.

The complete transformation process is depicted in Figure 9. It is divided into 5 individual
steps. Now let's walk through the transformation process step by step.

4.4.1 Parsing the source model

A source model is provided, in our case, Roster.zoom in Listing 2. Parsing involves reading
actual source code of the model, or the textual representation of model roster and splitting
it into understandable language symbols. This is made possible by ZOOM’s formally defined

syntax. A parser will parse the textual representation of the roster model and generate an internal abstract syntax tree (AST) representation of roster, which is an object tree. The resulting AST representation is shown in Figure 8.

4.4.2 Traversing the object tree

Once the metamodel is generated as AST, which is accessible by the transformation process through traversing the tree. We use an algorithm of “pre-order” traversal of the tree which was introduced in section 4.3. It means each node is visited before its children are visited and the root is visited first. The traversing is exemplified by pseudo code shown in Listing 1. since the matching metamodel node with rule(step 3) and generating target text(step 4) happens during the process of traversing. The pseudo code in Listing 1 actually shows all of these 3
steps. To trigger this transformation algorithm, the root node of the source metamodel needs to be passed. In our case, it is passing as \texttt{transformnode(m1, RuleSet, Output\_Model)}, since \texttt{m1} is the root node.

As we can see in Definition 4.2, a transformation rule has two mapping parts, \textit{T}_{pre} and \textit{T}_{post}. They are represented as \texttt{rule.pre} and \texttt{rule.post} in pseudo code of Listing 1. Shown in Listing 1: \texttt{rule.pre} is the mapping before traversing children of selected element, while \texttt{rule.post} is the mapping after traversing children of selected element. And as shown in line
5-7 in Listing 1, each node in the metamodel will be visited once and all its children nodes will get visited.

4.4.3 Matching node in object tree with rule

The key step in this transformation process is applying rules to source models represented by their metamodels. At this point, the transformation engine enters into the picture. As implied in the transformation rule definition, \( r = (P \rightarrow (T_{pre}, T_{post})) \), to apply a rule on a certain model includes both matching the pattern \( P \) and implement mapping \( T_{pre} \) and \( T_{post} \). The pattern \( P \) is specified to make sure that the correct node is being located and used. In the pseudo code in Listing 1, \( \text{rule.pre} \) (line 3) and \( \text{rule.post} \) (line 8) function as a template for the transformation. When implementing a template, we allow both static and dynamic specification. Static specification is a verbatim mapping and dynamic specification can be in one of these two forms: macros or JSP alike syntax.
4.4.4 Generating target text

The generation of target text can be as simple as outputting the text included in the rule \texttt{pre} or \texttt{post} elements as shown in Listing 1 line 3 and line 8. However, a more complicated scenario can be involved in this step. Figure 10 shows Step 3 and Step 4 in more details. The result of these steps is Literate Target Code. It will be used as input in the final step, post processing. In most of the cases, expressions are in different forms in source and target model. For example, a set addition expression in ZOOM specification: \texttt{a += b}, will need to be mapped as \texttt{a.add(b)} in Java syntax. We provide an extensible mechanism to assist the transformation, or mapping.

The construct in ZOOM that is similar to classes and interfaces in Java and C++ is \texttt{struct}. Structs are modeling constructs that define the structures, design requirements, and behaviors at a higher level of abstraction than classes and interfaces, which are primarily
implementation constructs. Structs can be considered as prototypes of classes and interfaces.

The construct in ZOOM that is similar to a method in Java is an operation. The syntax of operation in ZOOM also has close resemblance to methods in Java and C++ so that transformations between them involve mostly keywords mapping and structure rearrangement.

Most Java expressions are also legal in ZOOM. There are additional expressions supported by ZOOM such as integer expressions, logical expressions, set and list expressions [94]. For these additional expressions, we provide an automatic mechanism to transform them to expressions of target language. This mechanism is invoked whenever expressions are in different forms in the source and target model. Just like making the transformation template extensible, we design this external expression mapping mechanism so that extension is feasible and straightforward.

Figure 11 shows an example of mapping an expression $a += b$ into a target expression: $a.add(b)$. An expression mapping table is used for model transform engine to look up a corresponding expression in a target platform. This table is extensible in the sense that any specific mapping to a new target platform can be easily added and maintained.
Our external expression mapping is an extension of this template. It is a customized template for expression transformation. We define the expression mapping as:

**Definition 4.4:** An external expression mapping is defined as a 3-tuple: $M = (o, \lambda, \rho)$

- $o$ is the operator of expression
- $\lambda$ is the source expression template
- $\rho$ is the target expression template

The example shown in Figure 11 maps an expression $a += b$ into target expression $a.add(b)$. Using definition 6.1, it can be specified as:

$$M = (o, \lambda, \rho)$$
$$o = +=$$
$$\lambda = $v1 += $v2$$
$$\rho = $v1.add($v2)$$

A more visual way to represent the mapping definition is using a table such as Table 5.

The advantage of such an expression mapping schema lies in its extensibility. Expression mapping rules can be adequately represented by enumerating all possible target expressions. Also, different sets of rules can be compiled in such structural way for different target platforms.
4.4.5 Post processing

The Literate Target Code generated in Step 4 as shown in Figure 9 may or may not result in a desirable order that fits the target technical platform. For example, when the target platform is Java environment, an attribute node in a struct is always visited after the struct node, but the final code will require the target attribute code (declaration, initialization etc) to be present within the target class code. This kind of rearrangement of code is done by post processing. Post processing is responsible for rearranging the Literate Target Code in a desirable style that fits to the target technical platform. Here we treated the Literate Target Code as pieces of segment that can be flexibly rearranged. A post process goes through all these pieces and place each of them in the right places in the final models or code. This frees the model transformation engine from the details of contextual requirements of the target platform.

The post process rearranges the specification in a desirable style that fits to the target technical platform. This approach has a similar style as proposed in Knuth’s Literate Programming [112]. Literate programming is a methodology that combines a programming language with a documentation language, thereby making programs more robust, more portable, more easily maintained, and arguably more fun to write than programs that are written only in a high-level language. The main idea is to treat a program as a piece of literature, addressed to human beings rather than to a computer. The program is also viewed as a hypertext document, rather like the World Wide Web. Here we treated the generated model specification or code as segments that can be flexibly rearranged so that they confirm to the requirements of the target technical platform.

In our approach, adopting such a post processing mechanism frees the model transforma-
tion engine from the details of contextual requirements of the target platform. It makes the
development of transformation templates more straightforward.

After all the above transformation steps, the results are models or source code of the target
platform. In our example, the results are Java source code.
5 The HRMT Tool Suite

Chapter 4 introduces our model transformation approach and discusses the characteristics of this approach. We have successfully applied this approach in the implementation of the HRMT Tool. In this chapter, we will discuss the architecture of the HRMT tool and its main components.

5.1 HRMT Architecture

Figure 4 shows the basic structure of our model transformation framework. A transformation engine takes the HRM-defined source model as input and uses a template comprised of a set of transformation rules to produce an output model in a format specified by the templates. In Figure 12, the architecture of HRMT depicts the transformation engine in the shaded area. The main components such as ASTBuilder and ModelProcessor are shown in the rectangular box and the line between them represents interaction.

In Section 4.4, we show the complete transformation process in Figure 9. We also present a walk-through of the transformation process step by step. Here we will briefly show the steps using the main components shown in Figure 12.

- Parsing the source model

This step is performed by component ASTBuilder. Parsing involves reading the actual source code of the model, or the textual representation of the model roster, and splitting it into understandable language symbols. This is made possible by ZOOM’s formally defined syntax. ASTBuilder will parse the textual representation of a source model and will then generate an internal abstract syntax tree (AST) representation of it, which
is an object tree depicting metamodel.

- Traversing the object tree

Once a metamodel is generated as an AST, which is accessible by the ModelProcessor through traversing the tree, we use an algorithm of “pre-order” traversal of the tree which is introduced in section 4.3. This means each node is visited before its children nodes are visited and the root is visited first.

- Matching node in object tree with rule

The key step in this transformation process is applying rules to source models represented by their metamodels. TemplateProcessor is responsible for reading the template file and making available the rules to ModelProcessor. When ModelProcessor encounters a node in the object tree, it will use the path of the node to locate a corresponding rule in the template.
• Generating target text

The generation of target text can be as simple as outputing the text included in the rule pre or emphpost elements as shown in Listing 1 line 3 and line 8. However, a more complicated scenario can be involved in this step. First of all, for variables in a template whose value depends on a metamodel, they will need to be evaluated by Evaluator. In most of cases, expressions are in different forms in the source and the target model. For example, a set addition expression in ZOOM specification: \( a += b \), will need to be mapped as \( a.add(b) \) in Java syntax. This is the job of ExpressionMapper. We discuss the details of expression transformation in Chapter 6.

• Post processing

The text generated by ModelProcessor is called the Literate Target Code. It may or may not be in a desirable order that fits to the target technical platform. For example, when target platform is in a Java environment, an attribute node in a struct is always visited after the struct node, but the final code will require the target attribute code(declaration, initialization etc) to be present within the target class code. This kind of rearrangement of code is done by PostProcessor. PostProcessor is responsible for rearranging the Literate Target Code in a desirable style that fits to the target technical platform. Here we treated the Literate Target Code as pieces of a segment that can be flexibly rearranged. PostProcessor goes through all these pieces and place each of them in the right places in final models or code. This frees the ModelProcessor from the details of contextual requirements of the target platform.
5.2 HRMT Main Components

Figure 12 shows the main components of HRMT tool. In this section, we will provide further information about each of the components. Also since HRMT is an integral part of the ZOOM research project, some of the components, such as ASTBuilder and Evaluator are common components shared by other ZOOM tools as well. We will point out such shared components in the following discussion.

- **ASTBuilder**

ASTBuilder parses the textual representation of source model and generates an internal abstract syntax tree (AST) representation, which is an object tree depicting the metamodel. The object tree will be directly accessed by ModelProcessor. ASTBuilder is a common component used by other ZOOM tools such as ZOOM Animator and Event Based Framework. There are other smaller components that are shared by HRMT and other ZOOM Tools like TypeChecker. Here we only list those major ones that are critical to model transformation process.

- **TemplateProcessor**

TemplateProcessor is responsible for reading the template file and make available the rules to ModelProcessor. When ModelProcessor encounters a node in the object tree, it will use the path of the node to locate corresponding rule in template. Both Evaluator and ExpressionMapper are components that assist TemplateProcessor to provide mapping rule to ModelProcessor.

- **Evaluator**
The value of variables in template is determined by the metamodel that is being processed by ModelProcessor. Evaluator will provide this service of retrieving value. ASTBuilder is also a common component used by other ZOOM tools such as ZOOM Animator and Event Based Framework.

- **ModelProcessor**

ModelProcessor is the central component of HRMT. Once metamodel is generated as AST, ModelProcessor traverses the object tree, locates the corresponding rule and generates the Literate Target Code.

- **PostProcessor**

PostProcessor is responsible for rearranging the Literate Target Code in a desirable style that fits to the target technical platform. Here we treat the Literate Target Code as pieces of segment that can be flexibly rearranged. PostProcessor goes through all these pieces and places each of them in right places in final models or code. This frees the ModelProcessor from the details of contextual requirements of target platform.

- **ExpressionMapper**

In cases where expressions are in different forms in source and target model, ExpressionMapper is used to provide a mapping between these forms. Our external expression mapping is an extension of template. It is a customized template for expression transformation. The definition of expression mapping and example can be found in section 4.4.4.

The advantage of such an expression mapping schema lies in its extensibility. Expres-
sion mapping rules can be adequately represented by enumerating all possible target expressions. Also, different sets of rules can be compiled in such structural way for different target platforms.

5.3 Use of HRMT Tool Suite

To use the HRMT tool, a user needs to obtain ZOOM Software Suite first. We provide the latest ZOOM Installer in our research web site [113]. Documentation of how to use the HRMT is included.

After installation, when running the HRMT tool, a user has to provide the source model, the choice of template and the location of target files. Once this information is available, HRMT will automatically perform the transformation. When HRMT finishes the process, the user can go to the location of target files and check out all the files. The user will be allowed at this point to make changes to the target files, although it is unnecessary in most cases. A set of examples that include both the source model and template are provided. One simple way to get familiar with HRMT tool suite is for the user to take the example files, make modifications and then check the corresponding changes in target files. This process will make the transformation process more understandable.

Available templates are under a default directory called Template. Name of the available template is self-explanatory. For example, Template_NET.xml is a template for .NET platform. The default target directory is called Target. User will be allowed to change these default settings.
6 Development of Transformation

In section 4.4, while we introducing the transformation process we used an example of Community Sports League (CSL) roster management system. We will continue use the example in this section to show how transformation rules are constructed for transformation into basic classes (both Java and .NET), classes with associations (both Java and .NET), EJB Entities and a comprehensive EJB application. To facilitate the understanding of the construction of rules, we will use concise and friendly names to depict fragments of code, as first introduced in section 4.4. All the related rules and generated code can be found in the Appendix.

6.1 Basic Classes

The way ZOOM notation is designed makes its “struct” a similar counterpart of “class” in Java and .NET. This similarity in a way simplifies the transformation rules. We will show you how rules are composed to help generate basic classes with data members and related accessory methods (getXxx()/setXxx()).

The following is the rule to map “struct” to corresponding Java class.

\[
\text{BasicClassesR} = (\text{BasicClassesP} \rightarrow (\text{BasicClassesT}_{pre}, \text{BasicClassesT}_{post}))
\]

\[
\text{BasicClassesP} = \text{struct}
\]

\[
\text{BasicClassesT}_{pre} = (\text{File}, \text{Import}, \text{CON}_{\alpha}, \text{FIELD}_{\alpha}, \text{GetSet}_{\alpha})
\]

\[
\text{BasicClassesT}_{post} = ()
\]

BasicClassesT_{pre} is described above using a group of code fragment name. Table 6 shows the meaning of each of these names.

A code fragment placeholder is an anchor to reserve a place for more detailed code frag-
Table 6: Explanation of code fragment names

<table>
<thead>
<tr>
<th>Name</th>
<th>Code Fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>java file creation</td>
</tr>
<tr>
<td>Import</td>
<td>import list</td>
</tr>
<tr>
<td>CON$\alpha$</td>
<td>placeholder for class constructors</td>
</tr>
<tr>
<td>FIELD$\alpha$</td>
<td>placeholder for class member of fields</td>
</tr>
<tr>
<td>GetSet$\alpha$</td>
<td>placeholder for get and set methods</td>
</tr>
</tbody>
</table>

Table 7: Explanation of code fragment names

<table>
<thead>
<tr>
<th>Name</th>
<th>Code Fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CON$\beta$</td>
<td>class constructors</td>
</tr>
<tr>
<td>FIELD$\beta$</td>
<td>class member of fields</td>
</tr>
<tr>
<td>GetSet$\beta$</td>
<td>class member of get and set methods</td>
</tr>
</tbody>
</table>

ments. In this case, more code will be generated after processing a Variable node. Here is the rule for variable:

VariableR = (VariableP $\rightarrow$ (VariableT$\text{pre}$, VariableT$\text{post}$))

VariableP = struct/Declaration/VariableDeclarator

VariableT$\text{pre}$ = (CON$\beta$, FIELD$\beta$, GetSet$\beta$)

VariableT$\text{post}$ = ()

VariableP = struct/Declaration/VariableDeclarator gives the pattern to find the variable nodes. And Table 7 meaning of code fragment names.

A struct can have multiple variables. In the case of struct Player, it has id, name, position, salary. Using

$$\sum_{k=0}^{k=m} CF_k$$

to represent the concatenation of $m$ code fragment(CF), $m$ being the total number of corresponding code fragment. For example, in the class of Player, it has 4 class members, three class constructor code fragments will be concatenated. And the representation is:
\[ \text{CON}_\alpha = \sum_{k=0}^{k=4} \text{CON}_{\beta k} \]

Using this representation, the relationship between placeholder with subscript \( \alpha \) and the code fragment with subscript \( \beta \) can be shown as:

\[ \text{CON}_\alpha = \sum_{k=0}^{k=m} \text{CON}_{\beta k} \]
\[ \text{FIELD}_\alpha = \sum_{k=0}^{k=m} \text{FIELD}_{\beta k} \]
\[ \text{GetSet}_\alpha = \sum_{k=0}^{k=m} \text{GetSet}_{\beta k} \]

After the process, the resulting is 3 files. In case of Java: Player.java, Team.java and League.java. And in case of .NET: Player.cs, Team.cs and League.cs. You can find complete code of these Java files in Appendix A.1.

### 6.2 Classes with Associations

We have shown above that simple classes can be generated using ZOOM “struct”. As you may notice that in the PIM of Listing 2, there are associations defined. In ZOOM model associations are defined as relations or subtypes of relations, i.e., maps and injections. As shown in Listing 2 line 21-28, role names and multiplicities are specified as the attributes of the \(\text{@Association} \) annotation of the relation, which are \(\text{roleLeft}, \text{roleRight}, \text{multiplicityLeft} \) and \(\text{multiplicityRight} \). After \(\text{roster.zoom} \) is parsed, all these information will be available to access in metamodel.
Object-oriented programming languages do not contain syntax or semantics to express associations directly [114]. Instead, associations are implemented by a combination of classes, attributes and methods, depends on the specification of their multiplicity. For example, line 24-26 of Listing 2 show that there is a many-to-one between Team and League. In the target Java code, Team Class will have attribute with type League, and method getLeague and setLeague; League Class will have attribute with type List<Team> and method getAllTeam, addTeam and removeTeam. Using mLeft and mRight as shorthand for multiplicityLeft and multiplicityRight, the specification of rule is shown as:

\[
\text{AssociationR} = \ (\text{AssociationP} \rightarrow (\text{AssociationT}_{\text{pre}}, \text{AssociationT}_{\text{post}}))
\]

\[
\text{AssociationP} = \ \text{ZOOMModule/Declaration/VariableDeclarator}
\]

\[
\begin{align*}
\text{AssociationT}_{\text{pre}} = & \begin{cases} 
(\text{Att}_{\text{left}}, \text{FIELD}_{\text{right}}, \text{GetSet}_{\text{left}}, \text{GetSet}_{\text{right}}) & \text{if} (m_{\text{Left}} = \text{One and } m_{\text{Right}} = \text{One}) \\
(\text{Att}_{\text{left}}, \text{FIELD}_{\text{right}}, \text{GetSet}_{\text{left}}, V\text{GetSet}_{\text{right}}, AddRemove_{\text{right}}) & \text{if} (m_{\text{Left}} = \text{One and } m_{\text{Right}} = \text{Many}) \\
(V\text{Att}_{\text{left}}, \text{FIELD}_{\text{right}}, V\text{GetSet}_{\text{left}}, \text{GetSet}_{\text{right}}, AddRemove_{\text{left}}) & \text{if} (m_{\text{Left}} = \text{Many and } m_{\text{Right}} = \text{One}) \\
(V\text{Att}_{\text{left}}, \text{FIELD}_{\text{right}}, V\text{GetSet}_{\text{left}}, V\text{GetSet}_{\text{right}}, AddRemove_{\text{right}}, AddRemove_{\text{right}}) & \text{if} (m_{\text{Left}} = \text{Many and } m_{\text{Right}} = \text{Many})
\end{cases}
\]

\[
\text{AssociationT}_{\text{post}} = ()
\]

And the meaning of the code fragment names is shown in Table 8.
<table>
<thead>
<tr>
<th>Name</th>
<th>Code Fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIELD\textsubscript{left}</td>
<td>class member of left role field</td>
</tr>
<tr>
<td>FIELD\textsubscript{right}</td>
<td>class member of right role field</td>
</tr>
<tr>
<td>VFIELD\textsubscript{left}</td>
<td>class member of vector left role field</td>
</tr>
<tr>
<td>VFIELD\textsubscript{right}</td>
<td>class member of vector right role field</td>
</tr>
<tr>
<td>GetSet\textsubscript{left}</td>
<td>class member of left role get and set methods</td>
</tr>
<tr>
<td>GetSet\textsubscript{right}</td>
<td>class member of right role get and set methods</td>
</tr>
<tr>
<td>VGetSet\textsubscript{left}</td>
<td>class member of vector left role get and set methods</td>
</tr>
<tr>
<td>VGetSet\textsubscript{right}</td>
<td>class member of vector right role get and set methods</td>
</tr>
<tr>
<td>AddRemove\textsubscript{left}</td>
<td>class member of vector left role add and remove methods</td>
</tr>
<tr>
<td>AddRemove\textsubscript{right}</td>
<td>class member of vector right role add and remove methods</td>
</tr>
</tbody>
</table>

### 6.3 EJB Entities

Enterprise Java Bean (EJB) is a managed, server-sided component for modular construction of enterprise applications. The latest version of EJB is 3.0. In the EJB 3.0 world, all kinds of enterprise beans, including entity beans are just plain old Java objects (POJO) with appropriate annotations. The fact that ZOOM notation supports annotation makes the transformation between ZOOM model to EJB POJO fairly straightforward. Similarly to rules in 7.1 shows the rule for generation of Java classes, including construction, operation and attribution. The EJB 3.0 specification lets you specify O-R metadata via annotations. It provides a wide range of annotations that cover different types of relationships between POJOs, constraints, column information, sequence generators, composite primary key, and inheritance.

\[
\text{EJBEntityR} = (\text{EJBEntityP} \rightarrow (\text{EJBEntityT}_{\text{pre}}, \text{EJBEntityT}_{\text{post}}))
\]

\[
P = \text{struct}
\]

\[
\text{EJBEntityT}_{\text{pre}} = \text{(File, Import, Annotation, NamedQuery}_\alpha, \text{CON}_\alpha, \text{FIELD}_\alpha, \text{GetSet}_\alpha)
\]

\[
\text{EJBEntityT}_{\text{post}} = ()
\]
### Table 9: Explanation of code fragment names

<table>
<thead>
<tr>
<th>Name</th>
<th>Code Fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>java file creation</td>
</tr>
<tr>
<td>Import</td>
<td>import list</td>
</tr>
<tr>
<td>Annotation</td>
<td>annotations</td>
</tr>
<tr>
<td>NamedQuery&lt;α&gt;</td>
<td>placeholder for named queries</td>
</tr>
<tr>
<td>CON&lt;α&gt;</td>
<td>placeholder for class constructors</td>
</tr>
<tr>
<td>FIELD&lt;α&gt;</td>
<td>placeholder for class member of fields</td>
</tr>
<tr>
<td>GetSet&lt;α&gt;</td>
<td>placeholder for get and set methods</td>
</tr>
</tbody>
</table>

EJBEntityT<sup>pre</sup> is described above using a group of code fragment name. Table 9 the meaning of each of these names. And you can refer to Section 7.1 description for more details.

### 6.4 Comprehensive EJB Applications

Compared with the previous version of Enterprise Java Beans, the current generation of Java EE (EJB3) has several great improvements. Perhaps the most outstanding of these improvements is the new standard mechanism of EJB persistence, specified by JSR220. Such mechanism provides a transparent persistence layer based on radical reforms. These new features not only simplify the development process, while enhancing software reusability at the same time, but also make EJB3 more object oriented and model-driven friendly. For example, through elaborating mapping annotations, classes and relationships in a model can be directly mapped into database schema. For all that, the assignment of annotations and the code patterns for database manipulation still mean a lot of work for the programmer. Here we demonstrate how a comprehensive EJB3 application including entity beans and session beans can be transformed using our model.

The complete Roster EJB application incudes Java Persistence API entities or POJO
(Player, Team, and League), a stateful session bean (RequestBean), an application client (RosterClient). Except RosterClient, which is manually coded, all the entities and session beans are generated by the transformation process. We have attached in Appendix A.2 the complete rules and generated code for all of them.

We have shown the rule for generating POJO. The following are rules for session bean generation.

\[
EjbR = (EjbP \rightarrow (EjbT_{\text{pre}}, EjbT_{\text{post}}))
\]

\[
EjbP = \text{struct}
\]

\[
EjbT_{\text{pre}} = (\text{SessionInterfaceFile, SessionBeanFile, Import, Annotation, InterfaceMethod}_\alpha, \text{SessionMethod}_\alpha)
\]

\[
EjbT_{\text{post}} = ()
\]

\(EjbT_{\text{pre}}\) is described above using a group of code fragment names. Table 10 describing the meaning of each of these names.

The EJB 3.0 specification provides a Query API that can be used for both static and dynamic queries. A named query can be defined as a standalone query or attached to a query method of the bean class. You can define named queries in EJBQL or SQL. This is a boon for Java developers familiar with SQL syntax, as they can become EJB developers without having to learn another query language. Using our transformation framework, we can map operation specification in Listing 3 into named queries. The rule is described as following.

---

Table 10: Explanation of code fragment names

<table>
<thead>
<tr>
<th>Name</th>
<th>Code Fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SessionInterfaceFile</td>
<td>session interface java file creation</td>
</tr>
<tr>
<td>SessionBeanFile</td>
<td>session bean java file creation</td>
</tr>
<tr>
<td>Import</td>
<td>import list</td>
</tr>
<tr>
<td>Annotation</td>
<td>annotations</td>
</tr>
<tr>
<td>InterfaceMethod(\alpha)</td>
<td>placeholder for interface member of methods</td>
</tr>
<tr>
<td>SessionMethod(\alpha)</td>
<td>placeholder for session class member of methods</td>
</tr>
</tbody>
</table>
Table 11: Explanation of code fragment names

<table>
<thead>
<tr>
<th>Name</th>
<th>Code Fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SessionInterfaceMethod</td>
<td>interface member of method</td>
</tr>
<tr>
<td>SessionBeanMethod</td>
<td>session class member of method</td>
</tr>
<tr>
<td>NamedQuery_β</td>
<td>named query</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{EJBQuery}_R &= (\text{EJBQuery}_P \rightarrow (\text{EJBQuery}_T^{pre}, \text{EJBQuery}_T^{post})) \\
\text{EJBQuery}_P &= \text{struct/Operation} \\
\text{EJBQuery}_T^{pre} &= (\text{SessionInterfaceMethod}, \text{SessionBeanMethod}, \text{NamedQuery}_\beta) \\
\text{EJBQuery}_T^{post} &= ()
\end{align*}
\]

```java
@Session
struct Request {
    @query(entity=Player)
    Player findPlayer(String id) {
        return { Player p : p.id == id @ p };
    }

    @query(entity=Team)
    Team findTeam(String id) {
        return { Team t : t.id == id @ t };
    }

    @query(entity=Player)
    Set<Player> findAllPlayers() {
        return { Player p @ p };
    }

    @query(entity=Player)
    Set<Player> findByPosition(String position) {
        return { Player p : p.position == position @ p };
    }
}
```

Listing 3: Session Bean Specification

Again \( \text{EJBQuery}_T^{pre} \) is described above using a group of code fragment name. Table 11 shows the meaning of each of these names.

Here, the relation between place holder \( \text{NamedQuery}_\alpha \) and named query code fragment \( \text{NamedQuery}_\beta \) is:

\[
\text{NamedQuery}_\alpha = \sum_{k=0}^{k=m} \text{NamedQuery}_\beta^k
\]
And the result is shown in Listing 4

```java
@Entity
@Table(name="Player")

@NamedQueries ( {
    @NamedQuery(name = Player.findPlayer, 
    query = select p from Player p where p.id = :id),
    @NamedQuery(name = Player.findAllPlayers, 
    query = select p from Player p ),
    @NamedQuery(name = Player.findByPosition, 
    query = select p from Player p where p.position=:position) 
})
```

Listing 4: EJB Named Query
7 Case Study: A Hospital Information Management System

7.1 Introduction

In this chapter, we present a case study showing how our transformation framework can be used to transform a ZOOM model specification to applications running on multi-platforms. In order to show the power of our approach, the system described is not trivial. It is not a toy system, but it is a real-life example. This case study demonstrates how a PIM is transformed automatically into rather complex PSMs and code, and fulfills real-life needs. The complexity of the complete example is considerable. However, the example is not completely detailed out in all parts of the system in order to limit the size of this dissertation. In this chapter the requirements of the example system are stated in Section 7.2 and an overview is given of the models and transformations involved in Section 7.3. In Section 7.3, we point out the essential features of our approach that are illustrated in this case and elaborate on how these features are being illustrated in the case study.

7.2 Requirement of HIMS

7.2.1 Scope of HIMS

The Hospital Information Management System (HIMS) is a web application designed to improve access to patient information through a central electronic information system, an Electronic Healthcare Record (EHR) [115–117]. A HIMS’s goal is to streamline patient information flow and its accessibility for doctors and other health care providers. The im-
plementation of HIMS will improve patient care quality and patient safety over time.

Using MDE to develop a Healthcare System is an active research topic. Raistrick in [118] outlines how MDA and UML were used in the context of an extension of the processing of clinical data to provide a patient-based electronic record. In [119], a method was tested on a patient record of a hospital which provided rules for generating SGML/XML DTD element and parameter entity declarations from object-oriented UML class diagrams. In light of these research projects, we define the requirements of this case study in a similar complexity level compared to them.

7.2.2 Functionality Requirements

• Patient Registration

This component allows users to record, update or delete a new patient’s demographical information. The result will be saved in a centralized database.

• Hospital Personnel Management

This component allows users to record, update or delete hospital personnel, including doctors, nurses, staff member’s demographical information. The result will be saved in a centralized database.

• Appointments and Scheduling

This component allows users to record, update or delete an appointment between a patient and doctor. The result will be saved in a centralized database.

• Medical Reports

This component allows users to record, update, or delete a patient’s medical record.
These records need to support storing of heterogeneous data formats such as word documents or image files. The results will be saved in a centralized database.

7.2.3 Non-Functional Requirements

The support of non-functional requirements (NFR) by our approach is achieved in different ways depends on a specific requirement. We choose below some of the most important non-functional requirements in the scope of this case study. We have demonstrated how our framework can support customized solutions to a number of different NFRs. However, we would like to point out that this is far from a general solution to support all the different kinds of NFRs. Due to the wide range and vastly different characteristics of NFRs, solutions to address NFRs are heavily dependent on the target platform. A more comprehensive framework to address NFRs in the context of MDE remains a challenge for future work.

- **Security Requirement**

Putting EHR online benefits both healthcare providers and patients: each can access and utilize relevant information conveniently. However, putting the system on the open Internet also introduces new security concerns. Protecting privacy is fundamental to electronic health records and healthcare information exchange. In 2007, the Healthcare Information Technology Standards Panel [120] specified the technical standards needed to ensure the security of patient records. A reliable HIMS needs to address the security issue specifically and provides a security mechanism that is compatible with existing standards.

- **Scalability requirement**
Hospital information is an ever growing system. With the accumulation of healthcare records, this HIMS needs to be able to handle growing amounts of back end data and increasing data requests.

- **Performance requirement**

One of the key advantages of putting EHR online is ready access. This requires system response time in a reasonably quick fashion, especially with some data-intensive communications. This HIMS should have a mechanism to maintain desirable performance.

### 7.2.4 External Interface Requirements

The user interacts with the system using a web browser. Input from the user will be via keyboard input and mouse point and click. The user will navigate through the system by clicking on icons and links.

### 7.3 Modeling and Transformation of HIMS

Figure 13 is the ZOOM graphic representation for this case study. The texture representation in ZOOM can be found in Appendix A.5. This is the PIM of the software system. It is the only model that must be created by the model developer of the HIMS system.

The model in Figure 13 defines the information management system independently from any specific technology, so indeed, it is a PIM. But hospitals do not want a model; they want a running system. Therefore, we need to transform the PIM into a PSM that is compatible with the hospital’s technology infrastructure. In our case study, we choose Microsoft .NET and J2EE as the target web application platform, considering the popularity of both platforms. We also choose Microsoft Access and SQL Server as target database platforms. Appendix
7.4 Features Illustrated by Case Study

In this case study, we were able to showcase some of the main features of our approach. Each of these features will be elaborated upon.

7.4.1 Multi-platform support

The goal of this case study is to use the above model, or PIM and our model transformation framework to generate the final software system that can run on different platform. Since
this is a data-driven application, database system or platform plays an especially important part. We choose Microsoft .Net and J2EE as target web application platforms. We choose Microsoft Access and SQL Server as target database platform. In the end, we successfully transformed PIM into these four distinct target platforms.

- Microsoft .NET web application using Access as back end database
- Microsoft .NET web application using SQL Server as back end database
- J2EE web application using Access as back end database
- J2EE web application using SQL as back end database

### 7.4.2 Security requirement support

One of the most challenging problems in managing large web applications is the complexity of security administration and user-profile management [121]. Role-Based Access Control (RBAC) [122,123] has become the predominant model for advanced access control because it reduces the complexity and cost of administration. Under RBAC, security administration is greatly simplified by using roles, hierarchies and privileges. However, since different platforms provide different technical solutions to RBAC, it can be a daunting task to implement exactly the same kind of security requirements under different platforms. By presenting the example of automatically implementing the same RBAC security policy under different platforms based on the same model, we showcase the portability supported by our framework.

Both .Net and J2EE have their proprietary mechanism to support RBAC. They store the security information differently and require different coding routines to utilize the information. In our source model specification, we do not need to worry about such specific
Table 12: Example of ZOOM Annotation

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>@Security</td>
<td>specify whether role based access control will be enforced</td>
</tr>
<tr>
<td>@Table</td>
<td>specify whether a relational table will be created</td>
</tr>
<tr>
<td>@id</td>
<td>specify whether a primary key will be created</td>
</tr>
<tr>
<td>@restricted</td>
<td>specify whether querying a field will be restricted</td>
</tr>
</tbody>
</table>

proprietary mechanisms. We just need to specify that role-based access control will be enforced, as this line of specification shows:

@Security(Authentication="Yes", RoleManagement="Yes")

Also in our specification, to map a type of user group to a role, we just need to mark that struct and provide role name specification as shown in Listing 5.

As you can see, a couple more annotations are used in the specification. These annotations are used to communicate between the source model and the template so that particular part of template will be invoked. The use of annotations is similar to Java and EJB’s annotations. However, we make sure it is platform independent so that transformation templates for platforms other than Java and EJB can easily understand the annotations. Table 12 shows some of the annotations being used in the examples.

### 7.4.3 Scalability requirement support

Hospital information is an ever growing system. There are two main scenarios that an HIMS can grow. The first possible scenario refers to the increasing complexity of medical records, and the second one refers to the accumulation of medical records. We will discuss how our approach helps address both of these scenarios.

When medical records get more complicated, more properties need to be attached to a
namespace hospital;

@Security (Authentication = "Yes", RoleManagement = "Yes")

@table
struct Person {
    @id
    int id;
    String name;
    String address;
}

@table (role = "yes")
struct Patient extends Person{
    int id;
    String insurance;
}

Listing 5: Role Based Access Control Specification

record. This means the underlying data structure changes. From an implementation point of view, the change in data structure requires changing the database design and changing all the programs that utilize the data structure. It is a time consuming and error prone process to perform such changes, because more often than not some critical pieces of code will be missed in the updates. Using our approach, the only change required is to the source model. Making such changes is a time-consuming and error-prone process, because more often than not, some critical pieces of code will be missed in the updates. Using our approach, the only change required is to the PIM. By adding the new properties to the corresponding entities, a new running system can be generated again without worrying about missing code in the update.

The accumulation of medical records is best handled by a database system. If the amount of data collected eventually exceeds the limit of a database system, a more powerful database
system needs to be used instead. Again, in a software system that is developed in a traditional
process, potential updates must occur in all the places that perform database operations.
And of course, it is a time-consuming and error-prone process. Using our approach, in this
scenario, we do not even need to change the source model, because the underlying data
structure is not changed. We just need to update in the transformation templates the part
of the database connection that is being specified.

7.4.4 Performance requirement support

There are also two aspects of our case study that our approach can help support perfor-
mance requirements.

First as a data intensive application, such a HIMS’s performance depends on the database
system and database design. A good database design can effectively improve an application’s
performance. Since our support of generating SQL script directly from the source model , this
allows the modeler to change the database design in a modeling level and test the application
performance right away. Also, we can provide a template that generates XML files that can
be read by Hibernate and use Hibernate to map the source model to relational database.

This way, we can utilize Hibernate’s high performance object/relational persistence.

```
namespace hospital;

@table
struct MedicalDataTemplate {
    int id;
    String desc;
    String SecurityLevel;
    String format;
    @restricted
    String attachment;
}
```

Listing 6: Query Restriction
Second, as an online application, the performance is predicated upon the speed of the Internet. This means that an effective way to improve performance is to control the amount of data that is being transferred to the Intranet. In a traditional application development process, this task is performed ad hoc by individual pages. Using our approach, we can use annotation to communicate such restriction.

As shown in Listing 6 line 9, the annotation `restricted` will communicate the message to the template that unless specifically requested, the field attachment will not be queried automatically.
8 Related Work

Many contributions related to model transformation have been discussed in literature [57, 124, 125]. A number of solutions to describe and implement model transformation are currently available. Different top-level taxonomies can be found in [58], [126] and [127]. We base our classification mainly on [58] and the following is a list of work that has the closest relevance to our approach.

8.1 Direct-Manipulation Approach

Direct-manipulation approach offers an internal model representation plus some API to manipulate it. Tools using this approach are usually implemented as an object-oriented framework, which may also provide some minimal infrastructure to organize the transformations (e.g., abstract class for transformations). However, users have to implement transformation rules and scheduling mostly from scratch using a programming language such as Java. The direct-manipulation approach consists of providing some visitor mechanism to traverse the internal representation of a model and write code to a text stream. An example of this approach is Jamda [60], which is an object-oriented framework providing a set of classes to represent UML models, an API for manipulating models, and a visitor mechanism (so called CodeWriters) to generate code. Jamda does not support the MOF standard for defining new metamodels; however, new model element types can be introduced by subclassing the existing Java classes that represent the predefined model element types.

Jamda is an open-source framework for building application generators which create Java code from a model of the business domain. Instead of a generator which produces one
fixed architecture, Jamda provides a structure and building blocks so that you can build an application generator which does exactly what your project needs. It includes a sample generator for J2EE applications which can either be tailored to the needs of your J2EE project, or used as the basis of a generator for a completely different architecture.

From a UML model of the application domain, a generator created with Jamda could create the code for all the standard functions of locating, displaying and updating the business objects in the application. The developer would then concentrate on the application-specific business logic, which is merged into the generated application.

An application generator built using Jamda would perform the role of a model compiler in the Object Management Group’s Model Driven Architecture specification. It takes a UML domain model as input, adds new classes to the model to support the implementation, and then generates executable code.

The advantage of the Direct-Manipulation Approach is that it doesn’t require an additional mechanism to represent the metamodel, instead it traverses the internal representation directly. However, both [58] and [126] point out it is also its weakness because it requires user implement transformation rules and scheduling mostly from scratch using programming language such as Java. This makes the transformation process cumbersome and also makes reusing the transformation almost impossible.

8.2 Transformation using XSLT

Extensible Stylesheet Language Transformations (XSLT) is an XML-based language used for the transformation of XML documents into other XML. XSLT may be used effectively for some class of transformations of MOF models, as they may be represented as XML
documents via the XMI specification [128].

XMI is the basic language that helps to perform transformations among UML models as well as between UML models and other notations (especially code and other modelling languages). To realize XMI transformations it must be possible to get structured data out of the XMI documents. A promising approach is to use a standardized XML query or transformation language. There are several such languages [129] and the XSLT is the most widely used.

St-Denis et al. [130] compare various model interchange formats, e.g. RSF, XIF, XMI, and discuss the implementation details of an XMI-based model interchange engine. They identify the XMI’s support for differential model exchange as vital for the scalability, which is one of the requirements they use to assess the formats. Keienburg and Rausch [131] present an infrastructure for model evolution, schema migration and data instance migration, which is based on UML models. Successive differences on the evolution path are represented using the XMI’s differential elements. Yoda [131] presents an approach to developing applications using parameterize frameworks. The approach applies to the OMGs Model Driven Architecture (MDA). He recognizes model transformations as a way to customize predefined and parameterize frameworks.

Since models can be serialized as XML using XMI, implementing model transformations using XSLT, which is a standard technology for transforming XML, seems very attractive. Unfortunately, this approach has severe scalability limitations. Manual implementation of model transformations in XSLT quickly leads to non-maintainable implementations because of the verbosity and poor readability of XMI and XSLT. A solution to overcome this problem is to generate the XSLT rules from some more declarative rule descriptions, as demonstrated
in [132,133]. However, even this approach suffers from poor efficiency because of the copying required by the pass-by-value semantics of XSLT and the poor compactness of XMI.

8.3 Template-Based Approaches

Our approach belongs to Template-Based approaches, and the majority of currently available MDA tools support template-based model-to-code generation, e.g., JET, FUUT-je, Codagen Architect, AndroMDA, ArcStyler, OptimalJ and XDE (the latter two also provide model-to-model transformations). AndroMDA [134] reuses existing open-source template-based generation technology: Velocity [135] and XDoclet [136].

A template usually consists of the target text containing splices of metacode to access information from the source and to perform code selection and iterative expansion (see [137] for an introduction to template-based code generation). Template approaches usually offer user-defined scheduling in the internal form of calling a template from within another one. The logic accessing the source model may have different forms. The logic could be simply Java code accessing the API provided by the internal representation of the source model (e.g., JMI), or it could be declarative queries (e.g., in OCL or XPath [111]).

Compared to a Direct-Manipulation transformation, the structure of a template resembles more closely the code to be generated. Templates lend themselves to iterative development as they can be easily derived from examples. Since the template approaches discussed in this section operate on text, the patterns they contain are untyped and can represent syntactically or semantically incorrect code fragments. On the other hand, textual templates are independent of the target language and simplify the generation of any textual artifacts, including documentation.
In our evaluation chapter, we choose AndroMDA as an example. This is in consideration of AndroMDA being a typical transformation tools that uses template-based approach and the availability of AndroMDA documentations. AndroMDA is a code generation tool that takes a UML model as input and generates source code as output. It adopts a template-based transformation methodology similar to ours in a degree but differs significantly in handling of metamodel. Both have an extensible architecture consists of cartridges. These cartridges generate the code specific to a certain concrete technical platform. However, the fundamental difference between these two approaches are the metamodel that they base upon. AndroMDA uses MOF while we use HRM. Because of the complexity of the MOF compliant metamodel, when AndroMDA traverses its AST objects, it has to access them via a proprietary JMI interfaces, metamodel facades. Comparing to AndroMDA, our approach simplified the transformation template development by adopting a concise, tree-structure metamodel.

8.4 QVT(Queries/Views/Transformations)

Model transformation is a critical component of model-driven architectures (MDA). Recognizing this, a Request for proposal (RFP) was issued by OMG in 2002, on MOF Query/View/Transformation to seek a standard compatible with the MDA recommendation suite (UML, MOF, OCL, etc.) [19]. Several replies were provided by a number of companies and research institutions that evolved during three years to produce a common proposal that was submitted and approved [21–23].

Presently there are several products (commercial or open source) that claim compliance to the QVT standard. QVT defines a standard way to transform source models into target
models. There are several ideas in this proposal. One is that the source and target models may conform to arbitrary MOF metamodels. Another one is that the transformation program is considered itself as a model, and as a consequence also conforms to a MOF metamodel. This means more precisely that the abstract syntax of QVT should conform to a MOF 2.0 metamodel.

As a matter of fact, this is a bit more complex. First the QVT language integrates the OCL 2.0 standard and also extends it to imperative OCL. Second QVT defines not one but three domain-specific languages named Relations, Core and Operational Mappings and these languages are organized in a layered architecture. Relations and Core are declarative languages at two different levels of abstraction, with a normative mapping between them. The Relations language has a textual and a graphical concrete syntax. The QVT/OperationalMapping language is an imperative language that extends both QVT/Relations and QVT/Core. The syntax of the QVT/OperationalMappings language provides constructs commonly found in imperative languages (loops, conditions, etc.) [23].

Finally a mechanism called QVT/BlackBox for invoking transformation facilities expressed in other languages (XSLT, XQuery) is also an important part of the specification. It is especially useful for integrating existing non-QVT libraries and transformations.

For the time being the QVT standard only addresses model to model transformations, model meaning some entity conforming to any MOF 2.0 metamodel. All transformations of type model to text or text to model, whatever the text is (XML, Code, SQL, etc.), are presentely outside the scope of QVT and possibly subject to other standardization initiatives. They may be viewed as alternative transformation DSLs in the MDA technical space.

Duddy et. al. [138] propose a transformation language which will meet the requirements
of QVT RFP, and several others besides. The language is declarative and patterns based. Transformation descriptions are explicitly reusable and modular. Rules that make up such descriptions may be aspect-driven, allowing for transformations to be written to address semantic concepts rather than structural features. Peltier et al [62]. describe a transformation approach that operates on textual representations of models. The rules are expressed as a model instance, and then translated into a form that manipulates the textual documents. The current implementation uses XMI as the textual model format, and generates XSLT [139] from the model of the transformation rules which can then be applied to the XMI documents [22,140]. ATL(ATLAS Transformation Language) [141] is a model transformation language (MTL) developed by OBEO and INRIA to answer the QVT Request For Proposal. We choose ATL as an example for evaluation in the next chapter. Similar to the reason of choosing AndroMDA in template-based approach, this is in consideration of ATL being representative in this category and its availability of documentation.


9 Evaluation of Transformation Tools

Chapter 7 showcases what our approach can achieve as a MDE framework. Many of the features that the framework demonstrated are shared by some existing MDE tools. Chapter 8 lists a set of model transformation approaches that are closely related to our approach. However, what distinguishes our approach from others is the methodology that we adopted. The purpose of this chapter is to compare our model transformation approach with other tools to evaluate its strengths and weaknesses. This chapter is organized as follows. First, we introduce a set of metrics that we use to evaluate the transformation tools. Second, we discuss four individual tools that represent the four primary transformation approaches. These four tools will be compared against HRMR. Finally, we present the evaluation result.

As readability of metamodel and transformation definition is one of the advantages of our approach, we need to look deeper into the metrics that measure this quality. A large number of software product metrics have been proposed for the quality of software such as maintainability. Many of these metrics have not been properly validated due to poor methods of validation and non acceptance of metrics on scientific grounds [142,143]. In the literature, two types of validations, namely internal (theoretical) and external (empirical) are recommended [144]. Internal validation is a theoretical exercise that ensures that the metric is a proper numerical characterization of the property it claims to measure. Demonstrating that a metric measures what it purports to measure is a form of theoretical validation. External validation involves empirically demonstrating that a metric can be an important component or predictor of some software attributes of interest.

Kumar and Soni [145] have proposed a hierarchical model to evaluate qualities of object-
Table 13: Explanation of Metric Factor

<table>
<thead>
<tr>
<th>Metric</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Model Total Lines</td>
<td>number of lines in the text of source model</td>
</tr>
<tr>
<td>Source Model Total Tokens</td>
<td>number of tokens in the text of source model</td>
</tr>
<tr>
<td>Template Total Lines</td>
<td>number of lines in template</td>
</tr>
<tr>
<td>Template Model Total Tokens</td>
<td>number of tokens in template</td>
</tr>
<tr>
<td>Source Model Nesting Depth</td>
<td>deepest nesting level of source model</td>
</tr>
<tr>
<td>Template Model Nesting Depth</td>
<td>deepest nesting level of template</td>
</tr>
<tr>
<td>Cross Reference in Source Model</td>
<td>number of cross reference in source model</td>
</tr>
<tr>
<td>Inner Cross Reference in Source Model</td>
<td>number of inner cross reference in source model</td>
</tr>
<tr>
<td>Outer Cross Reference in Source Model</td>
<td>number of outer cross reference in source model</td>
</tr>
<tr>
<td>Cross Reference in Template</td>
<td>number of cross reference in template</td>
</tr>
<tr>
<td>Inner Cross Reference in Template</td>
<td>number of inner cross reference in template</td>
</tr>
<tr>
<td>Outer Cross Reference in Template</td>
<td>number of outer cross reference in template</td>
</tr>
<tr>
<td>Reference to Metamodel in Template</td>
<td>number of reference to metamodel in template</td>
</tr>
<tr>
<td>Unique Reference to Metamodel in Template</td>
<td>unique reference to metamodel in template</td>
</tr>
</tbody>
</table>

oriented software. This proposed model has been used for evaluation of maintainability assessment of object-oriented design quality, especially in design phase. In this model, quality factors such as maintainability are measured by a set of metrics such as Number of Classes(NOC), Number of Ancestors(NOA) and Number of Methods(NOM). In [146], they present empirical experiments to validate this hierarchical model of object-oriented design quality metrics. We will introduce a set of metrics that we identified for readability, shown in Table 13. We will explain what each metric means and the rationale of choosing it. Although we do not conduct individual validation of each metric, our choices of metrics are following the same practice demonstrated in Kumar and Soni’s study [145], and can be validated using a similar framework presented in [146].

We will further discuss these metrics by giving their definition and additional explanation.

- *Source Model Total Lines*(STotalLine)
Definition: The total line number of textualized source model excluding comments and empty lines.

The length of a document affects its readability directly. We measure total line number because it reflects the length of a document. In our evaluation, two kinds of textual source models are involved, ZOOM and XMI. Those metrics that are related to source model reflect the differences between ZOOM textual file and XMI files.

• Source Model Total Tokens ($S_{TotalToken}$)

Definition: The total token number of textualized source model excluding comments.

Two documents that have the same total line number don’t necessarily have the same length. The one with more tokens is a longer document. So total token number is a measurement of document length in addition to total line number. That is why we measure total token number of a document. As mentioned above, the comparison of source models are between ZOOM textual file and XMI files.

• Template Total Lines ($T_{TotalLine}$)

Definition: The total line number of textualized template excluding comments and empty lines.

All four tools except Jamda use template to represent transformation rules. Jamda uses Java API to directly manipulate metamodel. In Jamda’s case, the source code of Java API is the equivalence of template and will be evaluated the same way as other tools.

• Template Model Total Tokens ($T_{TotalToken}$)
Definition: The total line number of textualized template excluding comments.

As mentioned above, in the case of Jamda, Java API source code will be evaluated as template.

- **Source Model Nesting Depth (SNestDepth)**
  
  Definition: The depth of the deepest nested node in source model.
  
  Nesting depth reflects the levels of hierarchical document. The more levels a hierarchical document has, the harder it is for a reader to locate relevant information, which is why we choose it as an important factor of readability. For example, in the case of XMI, the depth of a node is clearly defined by XML standard. In other file format such as ZOOM textual file, we treat bracket "{" and "}" as the delimiter to count the depth of the deepest nested part of text.

- **Template Model Nesting Depth (TNestDepth)**
  
  Definition: The depth of the deepest nested node in template.
  
  As mentioned above, in the case of Jamda, Java API source code will be evaluated as template. And in case of textual files not in XML format in Listing 9, we treat bracket "{" and "}" as the delimiter to count the depth of the deepest nested part of text.

- **Cross Reference in Source Model (SCrossReference)**
  
  Definition: A cross reference in the source model is defined as a node or a part of text that explicitly uses different nodes or parts of text by referencing.
  
  We use the cross reference as an important factor for readability because a cross reference in a text requires a reader to locate information that is not adjacent to the current
text, which makes the document harder to read. We count all reference to nodes that
is not in the local node as cross references. For example, in the excerpt of XMI format
model in Listing 7, Line 12 is a cross reference to node in line 3. In this excerpt, both
Line 12 and 13 will be counted as cross references.

Listing 7: Excerpt from XMI model

• Inner Cross Reference in Source Model(SInnerCrossReference)

Definition: An inner cross reference in source model is defined as a node or a part
of text that explicitly use different nodes or parts of text within the source model by
referencing.

Inner cross reference is a subset of the cross reference. When a cross reference use only
the node or part of text within the source model, it is counted as inner cross reference.
For example, in the excerpt of XMI format model in Listing 7, Line 12 is an inner cross
reference to node in line 3.

• Outer Cross Reference in Source Model(SOuterCrossReference)
Definition: An outer cross reference in source model is defined as a node or a part of text that explicitly use different node or part of text outside the source model by referencing. Outer cross reference is a subset of the cross reference. When a cross reference use the node or part of text that is not included in the source model, it is counted as outer cross reference.

- **Cross Reference in Template(TCrossReference)**

  Definition: A cross reference in template is defined as a node or a part of text that explicitly use different node or part of text by referencing.

```
<xsl:template match="/">
  <xsl:for-each select="Content/Class">
    <xsl:call-template name="java-class">
      ...
    </xsl:call-template>
  </xsl:for-each>
</xsl:template>
...
<xsl:template name="java-class">
  ...
</xsl:template>
```

Listing 8: Excerpt from a XSLT template

Similar to cross reference in source model, we use the cross reference within template as important factors, because a cross reference in a text will require reader to locate information that is not adjacent to the current text. We count all reference to nodes that is not in the local node as cross references. For example, in the excerpt of XSLT template in Listing 8, Line 4 is a cross reference to node in line 10. In this excerpt, one cross reference will be counted.

- **Inner Cross Reference in Template(TInnerCrossReference)**
Definition: An inner cross reference in template is defined as a node or a part of text that explicitly use different node or part of text within the template by referencing.

Inner cross reference in template is a subset of cross reference in template. When a cross reference use only the node or part of text within the template, it is counted as inner cross reference. For example, in the excerpt of XSLT template in Listing 8, Line 4 is a inner cross reference to node in line 10.

• Outer Cross Reference in Template (TOuterCrossReference)

Definition: An outer cross reference in template is defined as a node or a part of text that explicitly use different node or part of text outside the template by referencing.

Outer cross reference in template is a subset of cross reference in template. When a cross reference use the node or part of text outside the template, it is counted as outer cross reference.

• Reference to Metamodel in Template (MetaReference)

Definition: A reference to metamodel in template is defined as any value inquiry to metamodel in the form of variable.

```
1 public $target.getterSetterTypeName ${target.getterName}() {
2     return this.${target.name};
3 }
4
5 public void ${target.setterName} ($target.getterSetterTypeName $target.name) {
6     this.${target.name} = ${target.name};
7 }
```

Listing 9: Excerpt from an AndroMDA template
Transformation templates provide the rules or instructions of how source model will be transferred. It is critical for template to reference to metamodel so that it can get the necessary information. This is accomplished by special variables that functions as placeholder for real value. Since more references to metamodel means additional steps to get real values from metamodel, this implies a more complicated template definition. That is why we choose this metric. For example, in excerpt of AndraMDA template in Listing 9, all variable start with $ is a reference to metamodel. In this excerpt, eight reference to metamodel will be counted.

- **Unique Reference to Metamodel in Template (UniqueMetaReference)**

Definition: A unique reference to metamodel in template is defined as any unique value inquiry to metamodel in the form of variable.

Unique reference to metamodel in template is a subset of reference to metamodel in template. For example, in excerpt of AndraMDA template in Listing 9, three appearance of ${target.name} will only be counted as one unique reference to metamodel.

In this excerpt, 4 unique references to metamodel will be counted.

In the remaining sections of this chapter, we choose one individual tool representing each transformation approaches that are discussed in Chapter 8. How transformation is accomplished in each of this tool is discussed. After that, an evaluation using above mentioned metrics will be presented and discussed.
9.1 Transformation Tools Being Evaluated

9.1.1 Direct-Manipulation Approaches: Jamda

Direct-manipulation approach consists in providing some visitor mechanism to traverse the internal representation of a model and write code to a text stream. An example of this approach is Jamda [60], which is an object-oriented framework providing a set of classes to represent UML models, an API for manipulating models, and a visitor mechanism (so-called CodeWriters) to generate code. Jamda does not support the MOF standard to define new metamodels; however, new model element types can be introduced by subclassing the existing Java classes that represent the predefined model element types.

Jamda takes a UML model as its input. This would usually be exported in XMI format by a case tool such as MagicDraw. To better demonstrate the difference between XMI and ZOOM notation as the source model, we use the Roster model described in Section 5, Figure 7.

For example, we show a fragment of the template that builds a file for each of the Java class tags in Listing 10.

Using Jamda to generate code involves a series of steps, including the followings:

- Define your system architecture

Before starting to generate code, the user must define what he or she wants it to produce, and also define the different layers, or tiers, in the system.

- Create new metamodel elements

For each of the kinds of classes or interfaces in the system architecture, it is necessary to
create a new Jamda metamodel class. The new class extends ClassType if it represents generated classes, or InterfaceType if it represents generated interfaces.

```
<uml:Model xmi:id='eee_1045467100313_135436_1'
            name='Data' visibility='public'>
  <xmi:Extension extender='MagicDraw UML 16.6 beta'>
    <moduleExtension ignoredInModule='true'/>
  </xmi:Extension>
</ownedComment>
<packageElement xmi:type='uml:Class'
                xmi:id='_16_6beta_1b480488_1252005357037_50629_501'
                name='Team' visibility='public'>
  <ownedAttribute xmi:type='uml:Property'
                  xmi:id='_16_6beta_1b480488_1252005383473_580357_519'
                  name='teamId' visibility='private'>
    <type xmi:type='uml:PrimitiveType'
          href='http://schema.omg.org/spec/UML/2.2/uml.xml#String'>
      <xmi:Extension extender='MagicDraw UML 16.6 beta'>
        <referenceExtension
          referentPath=UML Standard Profile::UML2 Metamodel
          ::AuxiliaryConstructs::PrimitiveTypes
          ::String::referentType='PrimitiveType'/>
      </type>
    </ownedAttribute>
</packageElement>
```

Listing 10: XMI Excerpt

- Create Model Transformers

The user needs to create one or more ModelTransformer implementations. The user could have one large ModelTransformer that carries out all the transformations, or divide it up into one for each kind of class to be generated.

- Create the configuration
Once the Model Transformer classes are created, the user needs to create a model compiler configuration to run them in the appropriate sequence. The final step will usually be to generate code by running the standard CodeGenTransformer.

As shown by these steps, when using Jamda much of the work that is done by our transformation framework must be done by users. For example, a Jamda user needs to create a metamodel class each time he uses the system, while in our framework, the metamodel is pre-defined in the system. And there is no extensible mechanism in Jamda.

9.1.2 Transformation using XSLT: Stylus Studio

Extensible Stylesheet Language Transformations (XSLT) is an XML-based language used for the transformation of XML documents into other XML document. XSLT may be used effectively for some class of transformations of MOF models, as they may be represented as XML documents via the XMI specification.

Stylus Studio is an advanced XML Integrated Development Environment (XML IDE) for building advanced XML applications. It supports using the XSLT processor to generate Java code. The XSLT processor ordinarily takes two input files - an XML source document, and an XSLT stylesheet. The XSLT stylesheet contains the XSLT program text and is itself an XML document that describes a collection of template rules: instructions and other hints that guide the processor toward the production of the output document. Since XSLT must be written in terms of the concepts in the source XMI document (model), and object (or element) creation explicit, the style is highly procedural and not easily extensible. Due to its XML basis, the concrete syntax is very user unfriendly. As such, it is unsuitable for one of the major goals of a declarative transformation language - which is to communicate

91
mapping specifications to human beings. It also requires a complete document as input, and is therefore not amenable to transforming incremental updates to models.

```xml
<xsl:template match="/"
  <xsl:for-each select="Content/Class">
    <xsl:variable name="filename" select="concat(’output/’, @name, ’Base.java’)" />
    <xsl:message>Creating
      <xsl:value-of select="$filename" /></xsl:message>
    <xsl:document href="{$filename}" method="text">
      <xsl:call-template name="java-class">
        <xsl:with-param name="class" select="." />
        <xsl:with-param name="associations" select="/Content" />
      </xsl:call-template>
    </xsl:document>
  </xsl:for-each>
</xsl:template>

<!-- The main template for a Java class -->
<xsl:template name="java-class">
  <xsl:param name="class" />
  <xsl:param name="associations" />
  /* This file has been generated */

  import java.util.*;

  public class <xsl:value-of select="$class/@name" > {

Listing 11: XSLT template for creating a file

For example, we show a fragment of a template that builds a file for each of the Java class using XSLT and HRMT approach in Listing 11 and Listing 12.

Compared to XSLT, our approach provides a shorter template language syntax. In most cases, transformation developer just need to fill in the syntax details of target platform when writing the specific template.
9.1.3 Template-Based Approaches: AndroMDA

AndroMDA is a code generation tool that takes a UML model as input and generates source code as output. It adopts a template-based transformation methodology similar to ours in a degree but differs significantly in handling of metamodel. Compared to direct-manipulation transformation, the structure of a template resembles more closely the code to be generated. Templates lend themselves to iterative development as they can be easily derived from examples. Since the template approaches discussed in this section operate on text, the patterns they contain are untyped and can represent syntactically or semantically incorrect code fragments. On the other hand, textual templates are independent of the target language and simplify the generation of any textual artifacts, including documentation.

Using a series of template files (which you can customize if you wish), AndroMDA can produce source code from a UML model in any programming language. Default templates exist to generate Java code (and in particular J2EE code) [134,147]. AndroMDA was designed to get the information necessary to generate code from MOF compliant models inside a MOF repository. On the other hand, at the heart of the code generator is a set of dy-
ynamic content templates, which can easily be replaced in order to support different target platforms. AndroMDA is based on Netbeans Metadata Repository (MDR) [148]. MDR is an implementation of OMG. Basically, MDR is able to load arbitrary models based on arbitrary metamodels, provided that you load the metamodel into MDR before you load the model.

```
package org.andromda.metafacades.uml14;

/**
 * Metaclass facade implementation.
 */
public class AttributeFacadeLogicImpl
    extends AttributeFacadeLogic
{
    public AttributeFacadeLogicImpl(
        org.omg.uml.foundation.core.Attribute metaObject,
        String context)
    {super(metaObject, context);
    }

    /**
     * @see org.andromda.metafacades.uml.AttributeFacade#getGetterName()
     */
    public java.lang.String handleGetGetterName()
    {
        return UMLMetafacadeUtils.getGetterPrefix(this.getType())
            + StringUtilsHelper.capitalize(this.getName());
    }

    /**
     * @see org.andromda.metafacades.uml.AttributeFacade#getSetterName()
     */
    public java.lang.String handleGetSetterName()
    {
        return "set" + StringUtils.capitalize(metaObject.getName());
    }
    ...
}
```

Listing 13: Metafacades example
MDR uses the MOF-to-Java mapping called JMI (Java Metadata Interface) [149], reads metamodels as XMI and generates JMI interface byte code for them. However, because of the intrinsic complexity of the MOF repository, AndroMDA uses Metafacades which are facades that are used to provide access to models loaded by a MDR repository. These “metafacades” provides a simpler access mechanism to the metamodel, but they also create one extra level of complexity that model transformation developers have to deal with.

Both AndroMDA and our approach are template-based, metamodel-based model transformation frameworks that support code generation. Both have an extensible architecture that consists of cartridges. These cartridges generate the code specific to a certain concrete technical platform. However, the fundamental difference between these two approaches are the metamodel that they are based upon. AndroMDA uses MOF while we use HRM.

Because of the complexity of the MOF compliant metamodel, when AndroMDA traverses its AST objects, it has to access them via proprietary JMI interfaces and metamodel facades. Compared to AndroMDA, our approach simplified the transformation template development
by adopting a concise, tree-structure metamodel.

Listing 14: Part of a typical metafacades descriptor

```xml
<metafacades>
  ...
  <property reference="arrayNamePrefix" default="ArrayOf"/>
  <property reference="schemaTypeMappingsUri"/>
  <metafacade class=org.andromda.cartridges.webservice.
    metafacades.WSDLTypeLogicImpl" contextRoot="true">
    <mapping class="org.omg.uml.foundation.core.Classifier$Impl"/>
  </metafacade>
  <metafacade class=org.andromda.cartridges.webservice.
    metafacades.WSDLTypeLogicImpl" contextRoot="true">
    <mapping class="org.omg.uml.foundation.core.UmlClass$Impl"/>
  </metafacade>
  <metafacade class=org.andromda.cartridges.webservice.
    metafacades.WSDLTypeLogicImpl" contextRoot="true">
    <mapping class="org.omg.uml.foundation.core.DataType$Impl"/>
  </metafacade>
  <metafacade class=org.andromda.cartridges.webservice.
    metafacades.WSDLEnumerationTypeLogicImpl>
    <mapping class="org.omg.uml.foundation.core.Classifier$Impl">
      <stereotype>ENUMERATION</stereotype>
    </mapping>
  </metafacade>
  <metafacade class=org.andromda.cartridges.webservice.
    metafacades.WSDLEnumerationTypeLogicImpl>
    <mapping class="org.omg.uml.foundation.core.Interface$Impl">
      <stereotype>ENUMERATION</stereotype>
    </mapping>
  </metafacade>
  ...
</metafacades>
```

Figure 14 shows the steps involved in model transformation in our approach and Andromda.

Writing metafacades is a non-trivial task. The user of a cartridge will load a model into AndroMDA and the metadata repository in AndroMDA will instantiates a metaobject for each element of the user’s model. After this, AndroMDA optionally instantiate a metafacade
object that shields the metaobject. After that, a metafacades developer can locate those
classes that end with "LogicImpl" in their names. These classes have been generated but
will never be touched by the generator again. This is the place where metafacades developers
implement their own logic.

Listing 13 shows an example of a LogicImpl class.

The metafacade descriptor allows the AndroMDA Core to discover a set of metafacades
on the classpath automatically. The AndroMDA Core also uses this descriptor to determine
what metafacades must be mapped to what meta model objects. The metafacades descriptor
must reside within the META-INF subdirectory of your cartridge (or shared metafacades
library) and must be named andromda-metafacades.xml. A typical metafacades descriptor
is shown in Listing 14.

9.1.4 QVT(Queries/Views/Transformations): ATL

Presently there are several products (commercial or open source) that claim compliance
to the QVT standard. QVT defines a standard way to transform source models into target
models.

ATL is a model transformation language (MTL) developed by OBEO and INRIA to answer
the QVT Request For Proposal. It can be used to do syntactic or semantic translation. ATL
is built on top of a model transformation Virtual Machine. A model-transformation-oriented
virtual machine has been defined and implemented to provide execution support for ATL
while maintaining a certain level of flexibility. As a matter of fact, ATL becomes executable
simply because a specific transformation from its metamodel to the virtual machine byte
code exists. Extending ATL is therefore mainly a matter of specifying the new language fea-
tures execution semantics in terms of simple instructions: basic actions on models (elements
creations and properties assignments).

---

```plaintext
module UML2JAVA;
create OUT : JAVA from IN : UML;

rule C2C {
  from e : UML!Class
  to out : JAVA!JavaClass (
    name <- e.name,
    isAbstract <- e.isAbstract,
    isPublic <- e.isPublic(),
    package <- e.namespace
  )
}

rule A2F {
  from e : UML!Attribute
  to out : JAVA!Field (
    name <- e.name,
    isStatic <- e.isStatic(),
    isPublic <- e.isPublic(),
    isFinal <- e.isFinal(),
    owner <- e.owner,
    type <- e.type
  )
}
```

Listing 15: ATL Rule example

Figure 15 shows the steps involved in model transformation in ATL. As shown in Figure 15,
the ATL approach takes model Ma.xmi as input and produces model Mb.xmi as output. Both
models may be expressed in the OMG XMI standard. The model Ma conforms to metamodel
MMa. Model Mb conforms to metamodel MMb. An ATL transformation template is
composed of a header, of a set of side-effect free functions called helpers and of a set of rules.

Since the abstract syntax of QVT conform to MOF 2.0 metamodel, one of the strengths
of our approach again is adopting a concise, tree-structure metamodel, HRM. Because of
the simplified metamodel, when transformation engine traverses its AST objects, it can have
direct access to the properties of the objects. This facilitates model transformer to develop
transformation template in a easier and quicker way. Secondly, for the time being the QVT
standard only addresses model to model transformations. All transformations of type model
to text, whatever the text is (XML, Code, SQL, etc.), are presently outside the scope of
QVT and possibly subject to other standardization initiatives. This is a restriction that
our approach doesn’t have, which make our approach more desirable in a code generation
scenario.

In the case of ATL, we can see the difference both from the input model and output model.
ATL use XMI in both input and output model, this makes the model specification much
more verbose than our approach. Also, ATL will require additional steps to accomplish code
generation. There is an associated ATL Development Toolkit plug in available in open source
from the GMT Eclipse Modeling Project (EMP) that implements the ATL transformation
Listing 15 shows an excerpt of ATL code for the transformation of a UML to Java consists of several functions and rules.

ATL helpers can be viewed as the ATL equivalence to Java methods. They make it possible to define factorized ATL code that can be called from different points of an ATL transformation. Listing 16 shows an excerpt of a typical helper specification. By organizing transformation rules in modules, ATL provides some extensibility. However it is a lower level of extensibility compared to AndroMDA and HRMT, where rules are organized in interchangeable units called cartridges.

### 9.2 Evaluation Result

After studying how these tools worked, we conducted an experimental trial on each of them. In the trial case, we used the CSL Roster example mentioned in Chapter 4. Using
PIM shown in Listing 2 and Figure 7, we generated Java code with each of the tools and evaluated the transformation using the metrics mentioned above.

9.2.1 Evaluation Result Reflecting Metamodel Complicity

Since all four tools use XMI as the format of the source model, the metrics evaluation is between the ZOOM input format and XMI format. We show the results in Table 14 and Figure 16. Additionally, because the choice of input model reflects essentially the choice of metamodel, it indeed reflects the complicity of HRM and MOF comparison.

We will discuss the evaluation result one by one:

- **SSTotalLine**

  SSTotalLine counts the total line number of textualized source model excluding comments and empty lines. The result in Table 14 shows that the total line number of XMI model that represents the same Roster model is about 8 times as much as the ZOOM model (266 vs. 33).

- **SSTotalToken**

  "Table 14: Evaluation Result of Source Model"

<table>
<thead>
<tr>
<th>Test Case</th>
<th>HRMT</th>
<th>Jamda/ Stylus/AndroMDA/ ATL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metamodel</td>
<td>HRMT</td>
<td>MOF</td>
</tr>
<tr>
<td>Input Format</td>
<td>ZOOM</td>
<td>XMI</td>
</tr>
<tr>
<td>SSTotalLine</td>
<td>33</td>
<td>266</td>
</tr>
<tr>
<td>SSTotalToken</td>
<td>67</td>
<td>1612</td>
</tr>
<tr>
<td>SNNesterDepth</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>SCrossReference</td>
<td>4</td>
<td>45</td>
</tr>
<tr>
<td>SInnerCrossReference</td>
<td>4</td>
<td>27</td>
</tr>
<tr>
<td>SOuterCrossReference</td>
<td>0</td>
<td>18</td>
</tr>
</tbody>
</table>
Figure 16: Evaluation Result of Source Model

STotalToken counts the total token number of textualized source model excluding comments. The result in Table 14 shows that the total token number of XMI model that represents the same Roster model is about 24 times of the ZOOM model (1612 vs. 67). Considering that XMI model has about 8 times the total number of lines, the average token number per line in XMI model is about 3 times as many as the ZOOM model.

- **SNestDepth**

SNestDepth counts the depth of the deepest nested node in the source model. The NestDepth of the XMI model is 8, as opposed to 2 of the ZOOM model. This difference reveals that it will be much harder to trace a node’s ancestors and descendants.

- **SCrossReference**

SCrossReference counts those references that point to places other than current con-
text. The result in Table 14 shows that the XMI model has a much higher SCrossReference number (45 vs. 4). A cross reference in a text will require the reader to locate information that is not adjacent to the current text. This result is a good indicator that ZOOM model is more readable in terms of avoiding cross references.

- **SInnerCrossReference**

  SInnerCrossReference counts only those SCrossReference that are within the source model. The result in Table 14 shows that all CrossReference in ZOOM is InnerCrossReference (4 out of 4) and 60 SInnerCrossReference number (27 vs. 4).

- **SOuterCrossReference**

  SOuterCrossReference counts only those SCrossReference that are outside the source model. The result in Table 14 shows that no CrossReference in ZOOM is InnerCrossReference (0 out of 4) and 40 to read additional document to fully understand the source model. It creates additional readability obstacle.

Overall the result in Table 14 shows that in all 6 metrics, ZOOM has significantly lower value compared to XMI. The STotalLine and STotalToken show that XMI model is much longer and verbose. The deeper nesting and significant amount of cross-references also made the XMI model harder to read. The result demonstrates that using HRM can significantly simplify the metamodel.

### 9.2.2 Evaluation Result Reflecting Transformation Complicity

The rest of the metrics are about the template. Since the template is the main document that needs to be developed in the transformation process, these metrics reflect the trans-
formation complexity. They are different from each other depending on the tools we are measuring. Table 15 and Figure 17 shows the result of comparing all the tools.

First of all, Jamda's metric measurement needs to be clarified. Since Jamda belongs to the Direct-Manipulation approach, it does not use a template like all the other tools do. So the measurement is taken from the Java API that did the transformation directly. The Java API in Jamda is treated as the equivalent of the template for measurement purpose. We will discuss the evaluation result one by one:

- **T_TotalLine**

  T_TotalLine counts the total line number of template excluding comments and empty lines. The result in Table 15 shows that HRMT has the lowest T_TotalLine number (44). In other words, HRMT has the shortest template in this test case.

- **T_TotalToken**

  T_TotalToken counts the total token number of template excluding comments. The result in Table 15 shows that HRMT has the lowest T_TotalToken number (108). In

<table>
<thead>
<tr>
<th>Test Case</th>
<th>HRMT</th>
<th>Jamda</th>
<th>Stylus</th>
<th>AndroMDA</th>
<th>ATL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate Java code for Roster</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T_TotalLine</td>
<td>44</td>
<td>652</td>
<td>82</td>
<td>207</td>
<td>70</td>
</tr>
<tr>
<td>T_TotalToken</td>
<td>108</td>
<td>1927</td>
<td>327</td>
<td>394</td>
<td>223</td>
</tr>
<tr>
<td>TNestDepth</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>T_CrossReference</td>
<td>2</td>
<td>37</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>T_InnerCrossReference</td>
<td>2</td>
<td>27</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>T_OuterCrossReference</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MetaReference</td>
<td>17</td>
<td>47</td>
<td>39</td>
<td>73</td>
<td>32</td>
</tr>
<tr>
<td>UniqueMetaReference</td>
<td>5</td>
<td>29</td>
<td>18</td>
<td>36</td>
<td>27</td>
</tr>
</tbody>
</table>

### Table 15: Evaluation Result of Template
other words HRMT is the least verbose in this test case,

- **TNestDepth**

TNestDepth counts the depth of the deepest nested node in template. The result in Table 15 shows that all five tools have similar TNestDepth(between 3-6).

- **TCrossReference**

TCrossReference counts those references that point to places other than current context. The result in Table 15 shows that except Jamda all other four tools have similar TCrossReference(between 2-4). Jamda has a significantly larger number of Cross-Reference(37). This shows that Direct-Manipulation approach will use much more CrossReference than other approaches.

- **TInnerCrossReference**
InnerCrossReference counts only those CrossReference that is within the template. The result in Table 15 shows that except Jamda all other four tools have similar CrossRef-erence (between 2-4). Jamda has a significantly larger number of CrossReference (27). This shows that Direct-Manipulation approach will use much more InnerCrossReference than other approaches.

- $T_{OuterCrossReference}$

$T_{OuterCrossReference}$ counts only those CrossReference that is outside the template. The result in Table 15 shows that except Jamda all other four tools don’t have $T_{OuterCrossReference}$. Jamda has a 10 $T_{OuterCrossReference}$. This shows that Direct-Manipulation approach make use of OuterCrossReference while other approach don’t.

- $MetaReference$

$MetaReference$ counts any value inquiry to metamodel in the form of variable. The result in Table 15 shows that HRMT has the smallest number (17). Some of the value inquiry is repetitive, so counting unique value inquiry or $UniqueMetaReference$ is necessary.

- $UniqueMetaReference$

$UniqueMetaReference$ counts unique value inquiry to metamodel in the form of variable. Again the result in Table 15 shows that HRMT has the smallest number (5). As accessing metamodel is the crucial step in model transformation, requiring less number of reference to metamodel reflects a simpler metamodel.

Overall in the case of $T_{TotalLine}$ and $T_{TotalToken}$, HRMT uses the shortest template
Table 16: Evaluatin Result of Transformation Tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>HRMT</th>
<th>Jamda</th>
<th>Stylus</th>
<th>AndroMDA</th>
<th>ATL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quality ↓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Readability of Metamodel</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Readability of Transformation Definition</td>
<td>High</td>
<td>Low</td>
<td>Medium</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>Extensibility</td>
<td>High</td>
<td>None</td>
<td>None</td>
<td>High</td>
<td>Medium</td>
</tr>
<tr>
<td>Availability of Supporting Tools</td>
<td>Low</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>High</td>
</tr>
</tbody>
</table>

compared to others. It is about half of ATL and only a fraction of Jamda, Stylus, and AndroMDA. Except for Jamda, there is no significant difference between nesting depth and cross-reference amount of all the approaches. This implies that all the tools except Jamda are used in a similar way to organize the template. The high number of CrossReference in Jamda is because it is using Java API to perform the transformation directly, and Java API organized their functions in different methods, files, and even in different packages. MetaReference and UniqueMetaReference are the most critical metrics because accessing metamodel information is the crucial step in model transformation. The more times that transformation has to access the metamodel, the more complicated the transformation process is. From the evaluation result, we can see that HRMT comes out using the least number of references to metamodel in both MetaReference and UniqueMetaReference. This is direct demonstration of having a simplified metamodel.

We recognize the limitation of this evaluation with data collected from only one case study. The results may vary to some extent when case studies of different scale and complexity are included. However, this limitation is partly mitigated by the fact the case study used is carefully chosen to represent a typical class of applications, and it is also used as the case
study by several other MDE tools.

It is necessary to point out that some of the lower values of metrics, namely total lines, tokens and meta references, are affected by HRMT being a lightweight approach. It raises the question of whether these values will go up if HRMT increases its functionalities. The answer to this question has everything to do with our fundamental design. The lightweight approach of using HRM for transformation will not change when adding additional functionalities. The two essential documents of transformationsource model and template, should not undergo any significant changes in further development. In other words, the trend of lower values of these metrics will be kept. However, it is possible that because of this lightweight design, HRMT may not accomplish all functionalities provided by an alternative approach, or to accomplish a particular functionality, HRMT has to be fundamentally changed. That scenario will present a good case study to delineate HRMT’s limitation.

The evaluation results for both set of metrics that we discuss above illustrate that HRMT provides a much shorter, less verbose and more readable source model and template. While evaluating the transformation tools in Section 9.1, we also compared the extensibility of each tool. What we found is that template-based transformation tools, namely AndraMDA and HRMT have a higher level of extensibility than ATL, while Jamda and Stylus Studio provide no extensible mechanism. However we are not claiming that HRMT exceeds in all aspects as a transformation tool. Using XMI as the format of the source model does give other tools some advantages over HRMT. The most obvious one being the availability of supporting tools such as editors. While the ZOOM notation HRMT based upon is still in research phrase, HRMT lacks such availability of supporting tools. Putting all these evaluation results together, Table 16 shows the final result of our evaluation.
10 Conclusion

10.1 Contributions

This dissertation provides a solution that uses a simplified metamodel as the foundation for building a template-based model transformation framework. This simplified metamodel is called the Hierarchical Relational Metamodel (HRM). The Hierarchical Relational Metamodel is built upon Z-based Object-Oriented Modeling notation (ZOOM). A template-based model transformation framework using Hierarchical Relational metamodel (HRM) is introduced. This framework aims to provide a simple, effective, and practical way to define model transformations. The benefits of this transformation framework include 1) readability and rigorousness of metamodel definitions; 2) simplicity of transformation definition; and 3) support of non-functional requirements.

The design of the Hierarchical Relational Metamodel (HRM) provides us a simplified way to understand and make use of the abstract syntax and semantics of the source models. This in turn helps the implementation of our transformation tool, the Hierarchical Relational Metamodel Transformation tool (HRMT). A case study is provided to demonstrate the validity and ability of HRMT. In this case study, a PIM represented by ZOOM notation is transformed into a full-fledged web application that can run on multiple platforms. Comparisons with related research work are also provided to show the benefits of this framework.

We would like to point out that while HRMT shares many of the benefits of MDE approaches, it suffers from some of the limitations as well. One of these limitations is the maintainability of the generated code. HRMT is a one-way transformation approach with no support for reverse engineering from the generated code. The maintainability problem
can be alleviated if modification to the generated code can be limited by making the changes in the models or the transformation instead. However, in reality, changes to the generated code is sometimes unavoidable. To fully address the maintainability of generated code will be a challenge for future work.

The main contribution of my research work is the design and realization of such a model transformation framework. It fulfills the MDE’s promise of Transforming PIM to PSMs for different platforms. However, many research projects and commercial tools can claim the same thing. What distinguishes our approach from others will most potently demonstrate the achievement of this research work. Such distinguished features of our approach are listed below.

- **Readability and rigorousness of metamodel definitions**

Metamodelling is a critical part of our model transformation approach. The Hierarchical Relational Metamodel (HRM) in our approach provides a mechanism to unambiguously define modeling languages - ZOOM in our case. The formal specification notation Z that ZOOM is based on ensures a mathematical rigorousness foundation. The tree structure among model elements that maintained by HRM provides a simplified way to read and make use of the abstract syntax and semantics of the source models.

- **Simplicity of transformation definition**

The design of the Hierarchical Relational Metamodel (HRM) provides us a simplified way to understand and make use of the abstract syntax and semantics of the source
models. The direct result of this is a simplified template definition in our approach. This is also demonstrated in the evaluation result in Chapter 9.

- **Support of non-functional requirements**

  The support of non-functional requirements by our approach is demonstrated by the case study discussed in Chapter 7.

  - **Support of Security Requirements**

    Security standards such as Role-Based Access Control (RBAC) are successfully adopted by modeling and model transformation.

  - **Support Performance Requirements**

    HRMT tool can be configured, through modeling, to control or improve the performance of a target software application.

  - **Support of Scalability Requirements**

    Depends on the specifics of scalability requirements, HRMT tool can be configured, through modeling or database script generation, to control the complexity of the transformation result.

### 10.2 Future Work

We conclude this dissertation by summarizing some future research topics.

- **Fine-tuning and optimizing the tool.**

  The case study in Chapter 7 demonstrates the validity of our approach and the ability of our tool. No major development of the tool will be necessary. However, there are
areas such as the efficient management of templates that can be fine tuned.

- **Integration with other tools.**

  As we pointed out in Section 2.3, one of the issues with XMI is its incompatibility with different tools. In order for our tool to gain better recognition, it is important that we provide a solution to the incompatibility issue in order to integrate our tool with other tools.

- **Interaction with other research fields.**

  While working on this dissertation, we came across a couple of research areas that have significant overlap with our research. These research areas, such as model checking and aspect-oriented programming, although not directly related to model transformation, may provide more potential research opportunities.
References


[22] DSTC CBOP and IBM. MOF Query/Views/Transformations, Revised Submission. OMG Document: ad/03-08-03.


122


126


127


A Development of Transformation

A.1 Generation of Basic Classes

Transformation Rule to Generate Basic Java Classes

```xml
<?xml version="1.0"?>
<!DOCTYPE RuleSet SYSTEM "rules.dtd">
<RuleSet name="simple" target="Java" postprocessor="xweb" import="stringfunction.zoom">
  <Rule match="ZOOMModule">
    <Pre><![CDATA[<?xml version="1.0"?>
<!DOCTYPE XWeb SYSTEM "xweb.dtd">
<XWeb>]]></Pre>
    <Post><![CDATA[
  </XWeb>]]></Post>
  </Rule>
  <Rule match="Struct">
    <Pre><![CDATA[
    <File name="${name}.java">
      import java.util.*;
      class ${name} {
        <MacroRef name="${name}.attrDef"/>
        <MacroRef name="${name}.constrDef"/>
        <MacroRef name="${name}.operDef"/>
      }
    </File>
    <MacroDef name="${name}.constrDef">
      ${name}{{<MacroRef name="${name}.constrAttrParamDef" />}{
        <MacroRef name="${name}.constrAttrDef"/>
      }}
    </MacroDef>
    <% if ((children != null) && (children.size() > 0)) { %>
      <MacroDef name="${name}.constrAttrParamDef">
        print(" ");
        for (int i=0; i<children.size(); i++) {
          if (children[i].elementName == "Declaration") {
            if (i == children.size() - 1) {
```
47    print(filterltgt(children[i].type.toString()));
48    print("\n");
49    print(children[i].name);
50    } else {
51    print(filterltgt(children[i].type.toString()));
52    print("\n");
53    print(children[i].name);
54    print(" ,\n");
55    }
56    }
57    }
58    }
59  
60  </MacroDef>
61 ]><?</Pre>  
62 </Rule>  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  

<Rule match="Struct/Declaration/VariableDeclarator">  
<Pre><![CDATA[  
<MacroDef name="{$parent.parent.name}.constrAttrDef">  
  this.${name} = ${name};  
</MacroDef>  
<MacroDef name="{$parent.parent.name}.attrDef">  
  protected ${filterltgt(parent.type.toString())} ${name};  
</MacroDef>  
<MacroDef name="{$parent.parent.name}.operDef">  
  public ${filterltgt(parent.type.toString())} get${capitalize(name.toString())}() {  
    return ${name};  
  }  
</MacroDef>  
<MacroDef name="{$parent.parent.name}.operDef">  
  public void set${capitalize(name.toString())}($${filterltgt(parent.type.toString())} ${name}) {  
    this.${name} = ${name};  
  }  
</MacroDef>  
]]></Pre>  
</Rule>  

</RuleSet>
import java.util.*;

class Player {
    protected String id;
    protected String name;
    protected String position;
    protected double salary;

    Player(String id, String name, String position, double salary)
    {
        this.id = id;
        this.name = name;
        this.position = position;
        this.salary = salary;
    }

    public String getId()
    {
        return id;
    }

    public void setId(String id)
    {
        this.id = id;
    }

    public String getName()
    {
        return name;
    }

    public void setName(String name)
    {
        this.name = name;
    }

    public String getPosition()
    {
        return position;
    }

    public void setPosition(String position)
    {
        this.position = position;
    }

    public double getSalary()
    {
        return salary;
    }
}
public void setSalary (double salary) {
    this.salary = salary;
}

}
import java.util.*;

class Team {
    protected String id;
    protected String name;
    protected String city;

    Team (String id, String name, String city) {
        this.id = id;
        this.name = name;
        this.city = city;
    }

    public String getId() {
        return id;
    }

    public void setId (String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName (String name) {
        this.name = name;
    }

    public String getCity() {
        return city;
    }

    public void setCity (String city) {
        this.city = city;
    }
}
League.java referenced in Section 6.1

```java
import java.util.*;

class League {
    protected String id;
    protected String name;
    protected String sport;

    League (String id, String name, String sport){
        this.id = id;
        this.name = name;
        this.sport = sport;
    }

    public String getId() {
        return id;
    }

    public void setId (String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName (String name) {
        this.name = name;
    }

    public String getSport() {
        return sport;
    }

    public void setSport (String sport) {
        this.sport = sport;
    }
}
```
Transformation Rule to Generate Basic .NET Classes

<?xml version="1.0"?>
<DOCTYPE RuleSet SYSTEM "rules.dtd">
<RuleSet name="simple" target=".NET" postprocessor="xweb" import="stringfunction.zoom">
    <Rule match="ZOOMModule">
        <Pre>
            <![CDATA[<?xml version="1.0"?>
<!DOCTYPE XWeb SYSTEM "xweb.dtd">
<XWeb>]]>
        </Pre>
        <Post><![CDATA[
        </Post>
    </Rule>
    <Rule match="Struct">
        <Pre><![CDATA[
using System;
class ${{name}} {
    <MacroRef name="${name}.attrDef"/>
    <MacroRef name="${name}.operDef"/>
}
]]>
    </Pre>
    <Rule match="Struct/Declaration/VariableDeclarator">
        <Pre><![CDATA[
private ${{filterltgt(parent.type.toString())}} ${name};
]]>
        </Pre>
        <MacroDef name="${parent.parent.name}.attrDef">
            public ${filterltgt(parent.type.toString())} ${capitalize(name.toString())}() {
                get { return ${name};}
                set { ${name} = value;}
            }
        </MacroDef>
    </Rule>
</RuleSet>
using System;

class Player {
    private string id;
    private string name;
    private string position;
    private double salary;

    public string Id() {
        get { return id; }
        set { id = value; }
    }

    public string Name() {
        get { return name; }
        set { name = value; }
    }

    public string Position() {
        get { return position; }
        set { position = value; }
    }

    public double Salary() {
        get { return salary; }
        set { salary = value; }
    }
}
using System;

class Team {
    private String id;
    private String name;
    private String city;

    public String Id() {
        get { return id; }
        set { id = value; }
    }

    public String Name() {
        get { return name; }
        set { name = value; }
    }

    public String City() {
        get { return city; }
        set { city = value; }
    }
}
using System;

class League {
    private string id;
    private string name;
    private string sport;

    public string Id() {
        get { return id; }
        set { id = value; }
    }

    public string Name() {
        get { return name; }
        set { name = value; }
    }

    public string Sport() {
        get { return sport; }
        set { sport = value; }
    }
}

League.cs referenced in Section 6.1
A.2 Generation of Classes with Associations

Transformation Rule to Generate Basic Java Classes with Associations

```xml
<?xml version="1.0"?>
<DOCTYPE RuleSet SYSTEM "rules.dtd">
  <RuleSet name="simple" target="Java" postprocessor="xweb" import="stringfunction.zoom">
    <Rule match="ZOOMModule">
      <Pre>
        <![CDATA[<?xml version="1.0"?>]]>
      </Pre>
      <Post><![CDATA[<DOCTY]]>
    </Rule>
  </RuleSet>
</DOCTYPE XWeb SYSTEM "xweb.dtd">
```

```java
class ${name} {
  <MacroRef name="${name}.attrDef"/>
  <MacroRef name="${name}.constrDef"/>
  <MacroRef name="${name}.operDef"/>
}

$${name} <MacroDef name="${name}.constrDef">${name} (<MacroRef name="${name}.constrAttrParamDef"/>) {
  <MacroRef name="${name}.constrAttrDef"/>
}</MacroDef>

<% if ((children != null) && (children.size() > 0)) { %>
<MacroDef name="${name}.constrAttrParamDef"%>
  print("");
  for (int i=0; i<children.size(); i++) {
    if (children[i].elementName == "Declaration") {
      if (i == children.size() - 1) {
        print(filterltgt(children[i].type.toString()));
        print(" ");
        print(children[i].name);
      } else {
```
    print(filterltgt(children[i].type.toString()));
    print("\n");
    print(children[i].name);
    print("\n");
})
}
%
</MacroDef>
]]>;/Pre>
</Rule>

<Rule match="Struct/Declaration/VariableDeclarator">
  <Pre><![CDATA[</Pre>
  <MacroDef name="${parent.parent.name}.constrAttrDef">
    this.${name} = ${name};
  </MacroDef>
  <MacroDef name="${parent.parent.name}.attrDef">
    protected ${filterltgt(parent.type.toString())} ${name};
  </MacroDef>
  <MacroDef name="${parent.parent.name}.operDef">
    public ${filterltgt(parent.type.toString())} get${capitalize(name.toString())}() {
      return ${name};
    }
  </MacroDef>
  <MacroDef name="${parent.parent.name}.operDef">
    public void set${capitalize(name.toString())}(${filterltgt(parent.type.toString())})
      (${parameter.toString()} }) {
      this.${name} = ${name};
    }
  </MacroDef>
  ]]>;/Pre>
</Rule>

<Rule match="Struct/Operation">
  <Pre><![CDATA[</Pre>
  <MacroDef name="${parent.name}.operDef">
    public ${filterltgt(type.toString())} ${name} (${parameter.toString()}) {
      <MacroRef name="${name}.exprDef"/>
    }
  </MacroDef>
  ]]>;/Pre>
</Rule>

<Rule match="Struct/Operation/BlockStatement/ExpressionStatement"
expressionmapping="yes">

```xml
<Pre><![CDATA[
<MacroDef name="${parent.parent.name}.exprDef">
]]>]]>[/Pre]
<Post><![CDATA[
</MacroDef>
]]>]]>[/Post]
</Rule>

<Rule match="ZOOMModule/Declaration/VariableDeclarator">
<Pre><![CDATA[
% if (parent.type != null && parent.type.gentype != null && parent.annotations != null && parent.annotations.size() > 0) {
%>
<MacroDef name="${parent.type.gentype[0]}.attrDef">
%>
% if (parent.annotations[0].multiplicityRight.toString() == "Multiplicity.Many") {
%>
protected List&<$parent.type.gentype[1]>&gt;$parent.annotations[0].roleRight;
%
</MacroDef>
<MacroDef name="${parent.type.gentype[0]}.operDef">
 public void add_$<parent.type.gentype[1]>
</MacroDef>

<MacroDef name="${parent.type.gentype[0]}.operDef">
 public void remove_$<parent.type.gentype[1]>
 ($<parent.type.gentype[1]>)$<parent.type.gentype[1]>; $parent.annotations[0].roleRight}.remove($<parent.type.gentype[1]>);
</MacroDef>

<MacroDef name="${parent.type.gentype[0]}.operDef">
 public void add_$<parent.type.gentype[0]>
 ($<parent.type.gentype[0]>)$<parent.type.gentype[0]>; $parent.annotations[0].roleLeft};
</MacroDef>
<MacroDef name="${parent.type.gentype[1]}.attrDef">
%>
% if (parent.annotations[0].multiplicityLeft.toString() == "Multiplicity.Many") {
%>
protected List&<$parent.type.gentype[0]>&gt;$parent.annotations[0].roleLeft;
%
</MacroDef>
<MacroDef name="${parent.type.gentype[1]}.operDef">
 public void add_$<parent.type.gentype[0]>
 ($<parent.type.gentype[0]>)$<parent.type.gentype[0]>; $parent.annotations[0].roleLeft}.add($<parent.type.gentype[0]>);
}
```
```xml
<MacroDef name="${parent.type.gentype[1]}.operDef">
  public void remove_${parent.type.gentype[0]} {
    $(parent.type.gentype[0]) ${parent.type.gentype[0]} }) {
    ${parent.annotations[0].roleLeft}.remove(${parent.type.gentype[0]});
  }
</MacroDef>

</MacroDef>

<%}
else {
  protected ${parent.type.gentype[0]} ${parent.annotations[0].roleLeft} ;
</MacroDef>

}%>
</Pre>
</Rule>
</RuleSet>
```
import java.util.*;

class Player {
    protected String id;
    protected String name;
    protected String position;
    protected double salary;
    protected List<Team> teams;

    Player (String id, String name, String position, double salary)
    {
        this.id = id;
        this.name = name;
        this.position = position;
        this.salary = salary;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getPosition() {
        return position;
    }

    public void setPosition(String position) {
        this.position = position;
    }
}
public double getSalary() {
    return salary;
}

public void setSalary(double salary) {
    this.salary = salary;
}

public void add_Team(Team Team) {
    teams.add(Team);
}

public void remove_Team(Team Team) {
    teams.remove(Team);
}
import java.util.*;

class Team {
    protected String id;
    protected String name;
    protected String city;

    protected List<Player> players;
    protected League league;

    Team (String id, String name, String city)
    {
        this.id = id;
        this.name = name;
        this.city = city;
    }

    public String getId () {
        return id;
    }

    public void setId (String id) {
        this.id = id;
    }

    public String getName () {
        return name;
    }

    public void setName (String name) {
        this.name = name;
    }

    public String getCity () {
        return city;
    }

    public void setCity (String city) {
        this.city = city;
    }

    public void add_Player (Player Player) {
        players.add(Player);
    }
}
public void remove_Player (Player Player) {
    players.remove(Player);
}
import java.util.*;

class League {
    protected String id;
    protected String name;
    protected String sport;

    protected List<Team> teams;

    League (String id, String name, String sport)
        {
            this.id = id;
            this.name = name;
            this.sport = sport;
        }

    public String getId () {
        return id;
    }

    public void setId (String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName (String name) {
        this.name = name;
    }

    public String getSport() {
        return sport;
    }

    public void setSport (String sport) {
        this.sport = sport;
    }

    public void add_Team (Team Team) {
        teams.add(Team);
    }

    public void remove_Team (Team Team) {
}
teams.remove(Team);
A.3 Generation of EJB Entities

Transformation Rule to Generate EJB Entities

```xml
<xml version="1.0"/>
<!DOCTYPE RuleSet SYSTEM "rules.dtd">
<RuleSet name="simple" target="EJB" postprocessor="xweb" import="stringfunction.zoom">
  <Rule match="ZOOMModule">
    <Pre><![CDATA[
    <xml version="1.0">
    <!DOCTYPE XWeb SYSTEM "xweb.dtd">
    <XWeb>
    ]]]]></Pre>
    <Post><![CDATA[
    </XWeb>]]>]]></Post>
    </Rule>

    <Rule match="Struct">
      <Mark>
      <Stereotype name="Entity"/>
      </Mark>
      <Pre><![CDATA[
      import org.{$parent.name}.entity.${name};
      <MacroDef name="{$parent.name}.createEntityInterface">
        public void create${name}(${name} ${decapitalize(name)});
      </MacroDef>
      <MacroDef name="{$parent.name}.createEntityOper">
        public void create${name}(${name} ${decapitalize(name)}) {
          em.persist(${name});
        }
      </MacroDef>
      <File name="${name}.java">
      package org.{$parent.name}.entity;
      import java.io.Serializable;
      import java.util.*;
      import javax.persistence.*;
      import static javax.persistence.FetchType.*;
      @Entity
      public class ${name} implements Serializable {
        <MacroRef name="${name}.attrDef"/>
        <MacroRef name="${name}.constrDef"/>
        <MacroRef name="${name}.operDef"/>
      }
    </File>
  </MacroDef>
  </MacroDef>
</File>
]]></Pre>
</Rule>

<MacroDef name="{$parent.name}.import">
  import org.{$parent.name}.entity.${name};
</MacroDef>

<MacroDef name="{$parent.name}.createEntityInterface">
  public void create${name}(${name} ${decapitalize(name)});
</MacroDef>

<MacroDef name="{$parent.name}.createEntityOper">
  public void create${name}(${name} ${decapitalize(name)}) {
    em.persist(${name});
  }
</MacroDef>
```

150
```java
${name} () {
    
    ${name} (<MacroRef name="${name}.constrAttrParamDef" />) {
        <MacroRef name="${name}.constrAttrDef"/>
    }
    
    </MacroDef>
}

//
if ((children != null) && (children.size()) > 0) {
    <MacroDef name="${name}.constrAttrParamDef">
        print(" ");
        for (int i=0; i<children.size(); i++)
        {
            if (children[i].elementName == "Declaration") {
                if (i == children.size() - 1) {
                    print(children[i]);
                } else {
                    print(children[i]);
                    print(" , ");
                }
            }
        }
    </MacroDef>
}]]</Pre>
</Rule>
</RuleSet>
```
package org.roster.entity;

import java.io.Serializable;
import java.util.*;
import javax.persistence.*;
import static javax.persistence.FetchType.*;

@Entity
public class Player implements Serializable {

  protected String id;
  protected String name;
  protected String position;
  protected double salary;

  Player () {
  }

  Player (String id, String name, String position, double salary) {
    this.id = id;
    this.name = name;
    this.position = position;
    this.salary = salary;
  }

  public String getId () {
    return id;
  }

  public void setId (String id) {
    this.id = id;
  }

  public String getName () {
    return name;
  }

  public void setName (String name) {
    this.name = name;
  }
}
public String getPosition() {
    return position;
}

public void setPosition(String position) {
    this.position = position;
}

public double getSalary() {
    return salary;
}

public void setSalary(double salary) {
    this.salary = salary;
}

package org.roster.entity;

import java.io.Serializable;
import java.util.*;
import javax.persistence.*;
import static javax.persistence.FetchType.*;

@Entity
public class Team implements Serializable {

    protected String id;
    protected String name;
    protected String city;

    Team () {
    }

    Team (String id, String name, String city)
    {
        this.id = id;
        this.name = name;
        this.city = city;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}
public String getCity() {
    return city;
}

public void setCity(String city) {
    this.city = city;
}
package org.roster.entity;

import java.io.Serializable;
import java.util.*;
import javax.persistence.*;
import static javax.persistence.FetchType.*;

@Entity
public class League implements Serializable {

    protected String id;
    protected String name;
    protected String sport;

    League () {
    }

    League (String id, String name, String sport)
    {
        this.id = id;
        this.name = name;
        this.sport = sport;
    }

    public String getId () {
        return id;
    }

    public void setId (String id) {
        this.id = id;
    }

    public String getName () {
        return name;
    }

    public void setName (String name) {
        this.name = name;
    }

    public String getSport () {
        return sport;
    }
}
public void setSport(String sport) {
    this.sport = sport;
}
A.4 Generation of Comprehensive EJB Applications

Transformation Rule to Generate Comprehensive EJB Applications

```xml
<?xml version="1.0"?>
<!DOCTYPE RuleSet SYSTEM "rules.dtd">

<RuleSet name="simple" target="EJB" postprocessor="xweb" import="stringfunction.zoom">
  <Pre>
    <! [CDATA[
    <Rule match="ZOOMModule">
      <Pre>
        <! [CDATA[
        <Rule match="Struct">
          <Mark>
            <Stereotype name="Entity"/>
          </Mark>
          <Pre>
            <! [CDATA[
            <MacroDef name="${parent.name}.import">
              import org.${parent.name}.entity.${name};
            </MacroDef>
            <MacroDef name="${parent.name}.createEntityInterface">
              public void create${name}(${name} ${decapitalize(name)});}
            </MacroDef>
            <MacroDef name="${parent.name}.createEntityOper">
              public void create${name}(${name} ${decapitalize(name)}) {
                em.persist(${name});
              }
            </MacroDef>
            <File name="${name}.java">
              package org.${parent.name}.entity;
              import java.io.Serializable;
              import java.util.*;
              import javax.persistence.*;
              import static javax.persistence.FetchType.*;
              @Entity
              @Table(name="${name}")
              @NamedQueries ( {
                <MacroRef name="${name}.queryDef"/>
              })
              public class ${name} implements Serializable {
                <MacroRef name="${name}.attrDef"/>
              }
```
<MacroRef name="${name}.constrDef"/>
<MacroRef name="${name}.operDef"/>
</File>

<MacroDef name="${name}.constrDef">
${name} () {
}

${name} (<MacroRef name="${name}.constrAttrParamDef" />) {
    <MacroRef name="${name}.constrAttrDef"/>
} </MacroDef>

<% if ((children != null) && (children.size() > 0)) {
    <MacroDef name="${name}.constrAttrParamDef">
        <%
        print("\n");
        for (int i=0; i<children.size(); i++) {
            if (children[i].elementName == "Declaration") {
                if (i == children.size() - 1) {
                    print(children[i]);
                } else {
                    print(children[i]);
                    print(",");
                }
            }
        }
    %>
    </MacroDef>
} %></Pre>
</Rule>

<Rule match="Struct">
    <Mark>
        <Stereotype name="Session"/>
    </Mark>
    <Pre><![CDATA[
        <File name="${name}.java">
            package org.${name}.session;
            import java.util.List;
            import javax.ejb.Remote;
            <MacroRef name="${name}.import"/>
            @Remote
            public interface ${name} {
                <MacroRef name="${name}.sessionInterface"/>
            }
        </File>
        <File name="${name}Bean.java">
            package org.${name}.session;
        </File>
    ]]>]]></Pre>
</Rule>
import java.util.List;
import javax.ejb.Remote;

@Remote
public class ${name}Bean implements ${name} {
  @PersistenceContext
  private EntityManager em;

  <MacroRef name="${name}.import"/>

  <Rule match="Struct/Operation">
    <Mark>
      <Stereotype name="query"/>
    </Mark>
    <Pre><![CDATA[
      String parameterString="";
      if ((children != null) && (children.size() > 0)) {
        for (int i=0; i<children.size(); i++)
        {
          if (children[i].elementName == "Parameter") {
            if (parameterString != "") parameterString += " , ");
            parameterString += children[i].toString();
          }
        }
      }
    </Pre>]]>
    <MacroDef name="${parent.name}.sessionInterface">
      public ${filterltgt(type.toString())} ${name}(${parameterString});
    </MacroDef>
    <MacroDef name="${parent.name}.sessionOper">
      public ${filterltgt(type.toString())} ${name}(${parameterString}) {
        <MacroRef name="${name}.exprDef"/>
      }
    </MacroDef>

    <%>
    if (this.annotations != null && this.annotations.size() > 0) {
      String entityname="";
      entityname=this.annotations[0].entity.toString();
    <%>
    </MacroDef>

    <%>
    if (this.annotations != null && this.annotations.size() > 0) {
      String entityname="";
      entityname=this.annotations[0].entity.toString();
    <%>
    </MacroDef>

    <%>

    @NamedQuery(name = "${entityname}.${name}"
    <MacroRef name=""name".namedqueryDef"/>

    <%>
    <MacroDef name=""name".queryDef" delimiter="," >

    <%>
    </MacroDef>

    <%>
    @NamedQuery(name = "${entityname}.${name}"
    <MacroRef name=""name".namedqueryDef"/>

    <%>
    </MacroDef>

    <%>
String parameterString = "";
if ((children != null) && (children.size() > 0)) {
    for (int i = 0; i < children.size(); i++) {
        if (children[i].elementName == "Parameter") {
            if (parameterString != "") parameterString += ", ";
            parameterString += children[i].toString();
        }
    }
    parameterString = "";
}

String var = "";
String exp = "";
String dec = "";
if ((children != null) && (children.size() > 0)) {
    for (int i = 0; i < children.size(); i++) {
        if (children[i].elementName == "Declaration") {
            dec = children[i].toString();
        }
        if (children[i].elementName == "BinaryExpression") {
            exp = " where " + children[i].leftOperand + " = :" + children[i].rightOperand;
        }
        if (children[i].elementName == "Name") {
            var = children[i].toString();
        }
    }
}
query = "select $\{var\}.from $\{dec\}.$\{exp\}"

```
query = "select $\{var\}.from $\{dec\}.$\{exp\}"
```

```java
String optype = ";
optype = parent.parent.parent.type.toString();
optype = filterltgt(optype);
return (${optype})
em.createNamedQuery("${parent.parent.parent.name}" ).getResultList();
```

```java
if (parent.type != null && parent.type.gentype != null && parent.annotations != null && parent.annotations.size() > 0) {
if ((parent.annotations[0].multiplicityRight.toString() == "Multiplicity.One") &&
    (parent.annotations[0].multiplicityLeft.toString() == "Multiplicity.One")) {
```
public $ {parent.type.gentype[1]}
get$ {capitalize (parent.annotations[0].roleRight.toString () )} () {
    return $ {parent.annotations[0].roleRight} ;
}
</MacroDef>

<MacroDef name="${parent.type.gentype[0]}.operDef">
public void set$ {capitalize (parent.annotations[0].roleRight.toString () )}
    ( ${parent.type.gentype[1]} $ {parent.annotations[0].roleRight} ) { 
    this.$ {parent.annotations[0].roleRight} = $ {parent.annotations[0].roleRight} ;
}
</MacroDef>

<MacroDef name="${parent.type.gentype[0]}.attrDef">
protected $ {parent.type.gentype[0]} $ {parent.annotations[0].roleLeft} ;
</MacroDef>

<MacroDef name="${parent.type.gentype[1]}.operDef">
@if ( (parent.annotations[0].multiplicityRight.toString () == "Multiplicity.One") &&
      (parent.annotations[0].multiplicityLeft.toString () == "Multiplicity.Many" ) ) {
    <--- Right:One ; Left: Many---
</MacroDef>
protected $\{\text{parent.type.gentype[1]}\} \$\{\text{parent.annotations[0].roleRight}\}$;
</MacroDef>

public $\{\text{parent.type.gentype[1]}\}$
    get$\{\text{capitalize(parent.annotations[0].roleRight.toString())}\]() {
        return $\{\text{parent.annotations[0].roleRight}\}$;
    }
</MacroDef>

protected List&l t ;$\{\text{parent.type.gentype[1]}\}&gt; $\{\text{parent.annotations[0].roleLeft}\}$;
</MacroDef>

if ((parent.annotations[0].multiplicityRight.toString() == "Multiplicity.Many") &
    (parent.annotations[0].multiplicityLeft.toString() == "Multiplicity.One")
    )
%
</MacroDef>

protected List&l t ;$\{\text{parent.type.gentype[1]}\}&gt; $\{\text{parent.annotations[0].roleRight}\}$;
</MacroDef>

protected List&l t ;$\{\text{parent.type.gentype[0]}\}&gt; $\{\text{parent.annotations[0].roleLeft}\}$;
</MacroDef>

@ManyToOne
public $\{\text{parent.type.gentype[1]}\}$
    get$\{\text{capitalize(parent.annotations[0].roleRight.toString())}\]() {
        return $\{\text{parent.annotations[0].roleRight}\}$;
    }
</MacroDef>

@ManyToOne
public $\{\text{parent.type.gentype[1]}\}$
    get$\{\text{capitalize(parent.annotations[0].roleRight.toString())}\]() {
        return $\{\text{parent.annotations[0].roleRight}\}$;
    }
</MacroDef>

@ManyToOne
public $\{\text{parent.type.gentype[1]}\}$
    get$\{\text{capitalize(parent.annotations[0].roleRight.toString())}\]() {
        return $\{\text{parent.annotations[0].roleRight}\}$;
    }
</MacroDef>

@ManyToOne
public $\{\text{parent.type.gentype[1]}\}$
    get$\{\text{capitalize(parent.annotations[0].roleRight.toString())}\]() {
        return $\{\text{parent.annotations[0].roleRight}\}$;
    }
</MacroDef>

%-- Entity table annotation --%
@ManyToOne
public $\{\text{parent.type.gentype[1]}\}$
    get$\{\text{capitalize(parent.annotations[0].roleRight.toString())}\]() {
        return $\{\text{parent.annotations[0].roleRight}\}$;
    }
</MacroDef>

%-- Entity table annotation --%
@ManyToOne
public $\{\text{parent.type.gentype[1]}\}$
    get$\{\text{capitalize(parent.annotations[0].roleRight.toString())}\]() {
        return $\{\text{parent.annotations[0].roleRight}\}$;
    }
</MacroDef>

%-- Entity table annotation --%
@ManyToOne
public $\{\text{parent.type.gentype[1]}\}$
    get$\{\text{capitalize(parent.annotations[0].roleRight.toString())}\]() {
        return $\{\text{parent.annotations[0].roleRight}\}$;
    }
</MacroDef>
public List&lt;$ {parent.type.gentype[1]}&gt; get()
{
    capitalize (parent.annotations[0].roleRight.toString())
    (List&lt;$ {parent.type.gentype[1]}&gt; $ {parent.annotations[0].roleRight})
    this.$ {parent.annotations[0].roleRight} = $ {parent.annotations[0].roleRight};
}

public void set(capitalize (parent.annotations[0].roleRight.toString()))
(
    $ {parent.type.gentype[1]} $ {parent.annotations[0].roleRight})
{
    this.$ {parent.annotations[0].roleRight} = $ {parent.annotations[0].roleRight};
}

protected $ {parent.type.gentype[0]} $ {parent.annotations[0].roleLeft};

public void set(capitalize (parent.annotations[0].roleLeft.toString()))
(
    $ {parent.type.gentype[0]} $ {parent.annotations[0].roleLeft})
{
    this.$ {parent.annotations[0].roleLeft} = $ {parent.annotations[0].roleLeft};
}

if ( (parent.annotations[0].multiplicityRight.toString() == "Multiplicity.Many") &&
    (parent.annotations[0].multiplicityLeft.toString() == "Multiplicity.Many") )
{
    (List&lt;$ {parent.type.gentype[1]}&gt; $ {parent.type.gentype[1]})
    return $ {parent.annotations[0].roleRight};
}

protected List&lt;$ {parent.type.gentype[1]}&gt; $ {parent.annotations[0].roleRight};

public void set(capitalize (parent.annotations[0].roleRight.toString()))
(
    $ {parent.type.gentype[1]} $ {parent.annotations[0].roleRight})
{
    this.$ {parent.annotations[0].roleRight} = $ {parent.annotations[0].roleRight};
}

@ManyToMany
@ManyToMany(mappedBy="$ {parent.annotations[0].roleLeft}
" , fetch=EAGER)
public List&lt;$ {parent.type.gentype[1]}&gt; get()
{
    capitalize (parent.annotations[0].roleRight.toString())
    return $ {parent.annotations[0].roleRight};
}
```java
protected List&lt;{parent.type.gentype[0]}&gt; {parent.annotations[0].roleLeft}=
{parent.annotations[0].roleRight};

&lt;/MacroDef&gt;

&lt;MacroDef name="{parent.type.gentype[1]}.attrDef"&gt;
protected List&lt;{parent.type.gentype[0]}&gt; ${parent.annotations[0].roleLeft} =
{parent.type.gentype[1]}.attrDef
&lt;/MacroDef&gt;

&lt;MacroDef name="{parent.type.gentype[1]}.roleDef"&gt;
public List&lt;{parent.type.gentype[0]}&gt; ${parent.annotations[0].roleRight} =
{parent.type.gentype[0]}.roleDef
&lt;/MacroDef&gt;

&lt;MacroDef name="{parent.type.gentype[1]}.roleDef"&gt;
public List&lt;{parent.type.gentype[0]}&gt;&lt;/MacroDef&gt;

&amp;lt;-- Entity table annotation --&amp;gt;
@ManyToOne
@JoinTable(name="{parent.type.gentype[1]}.{parent.type.gentype[0]}.ID",
joinColumns=@JoinColumn(name="{parent.type.gentype[1]}.ID",
inverseJoinColumns=@JoinColumn(name="{parent.type.gentype[0]}.ID",
referencedColumnName="
public List&amp;lt;{parent.type.gentype[0]}&gt;&lt;/MacroDef&gt;

&lt;MacroDef name="{parent.type.gentype[1]}.roleDef"&gt;
public List&amp;lt;{parent.type.gentype[0]}&gt; ${parent.annotations[0].roleLeft} =
{parent.type.gentype[0]}.roleDef
&lt;/MacroDef&gt;

&amp;lt;-- Entity table annotation --&amp;gt;
@ManyToMany
public List&amp;lt;{parent.type.gentype[0]}&gt;&lt;/MacroDef&gt;

&amp;lt;-- Entity table annotation --&amp;gt;
@ManyToMany
public List&amp;lt;{parent.type.gentype[0]}&gt;&lt;/MacroDef&gt;

&amp;lt;-- Entity table annotation --&amp;gt;
@ManyToMany
public List&amp;lt;{parent.type.gentype[0]}&gt;&lt;/MacroDef&gt;
```

166
<MacroDef name="\${parent.parent.name}.attrDef">
protected \${parent.type.name} \${name};
</MacroDef>

<MacroDef name="\${parent.parent.name}.operDef">
  if (name.toString() == "id") {
    print("@Id");
  }
</MacroDef>

@Column(name = "\${name}")
public \${parent.type} get\${capitalize(name.toString())}() {
  return \${name};
}
</MacroDef>

public void set\${capitalize(name.toString())}($\{parent.type\} $\{name\}) {
  this.$\{name\} = $\{name\};
}
</MacroDef>

</RuleSet>
package org.roster.entity;

import java.io.Serializable;
import java.util.*;
import javax.persistence.*;
import static javax.persistence.FetchType.*;

@Entity
@Table(name="Player")
@NamedQueries({

    @NamedQuery(name = "Player.findPlayer",
            query = "select p from Player p where p.id = :id"
    )

, @NamedQuery(name = "Player.findAllPlayers",
            query = "select p from Player p"
    )

, @NamedQuery(name = "Player.findByPosition",
            query = "select p from Player p where p.position = :position"
    )
})

public class Player implements Serializable {
    Player () {
    }

    Player (String id, String name, String position, double salary)
    {
        this.id = id;
        this.name = name;
        this.position = position;
        this.salary = salary;
    }
}
@Id
@Column(name = "id")
public String getId() {
    return id;
}

public void setId(String id) {
    this.id = id;
}

@Column(name = "name")
public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

@Column(name = "position")
public String getPosition() {
    return position;
}

public void setPosition(String position) {
    this.position = position;
}

@Column(name = "salary")
public double getSalary() {
    return salary;
}

public void setSalary(double salary) {
    this.salary = salary;
}

@ManyToOne
@ManyToMany(mappedBy="players", fetch=EAGER)
public List<Team> getTeams() {
    return teams;
}

public void setTeams(List<Team> teams) {
    this.teams = teams;
}

protected String id;

protected String name;
protected String position;

protected double salary;

protected List<Team> teams;
package org.roster.entity;

import java.io.Serializable;
import java.util.*;
import javax.persistence.*;
import static javax.persistence.FetchType.*;

@Entity
@Table(name="Team")
@NamedQueries({
  @NamedQuery(name = "Team.findTeam",
  query = "select t from Team t where t.id = :id")
})

public class Team implements Serializable {
  Team () {
  }

  Team (String id, String name, String city) {
    this.id = id;
    this.name = name;
    this.city = city;
  }

  @Id
  @Column(name = "id")
  public String getId () {
    return id;
  }

  public void setId (String id) {
    this.id = id;
  }

  @Column(name = "name")
  public String getName () {
    return name;
  }

  public void setName (String name) {
    this.name = name;
  }
}
@Column(name = "city")
public String getCity() {
    return city;
}

public void setCity(String city) {
    this.city = city;
}

@ManyToOne
public League getLeague() {
    return league;
}

public void setLeague(League league) {
    this.league = league;
}

protected String id;
protected String name;
protected String city;
protected List<Player> players;
protected League league;
package org.roster.entity;

import java.io.Serializable;
import java.util.
import javax.persistence.
import static javax.persistence.FetchType.*;

@Entity
@Table(name="League")
@NamedQueries ({})

public class League implements Serializable {
    League () {
    }

    League (String id , String name , String sport)
    {
        this.id = id;
        this.name = name;
        this.sport = sport;
    }

    @Id
    @Column(name ="id")
    public String getId () {
        return id;
    }

    public void setId (String id) {
        this.id = id;
    }

    @Column(name ="name")
    public String getName () {
        return name;
    }

    public void setName (String name) {
        this.name = name;
    }

    @Column(name ="sport")

    @Column(name = "id")
    public String getId () {
        return id;
    }

    public void setId (String id) {
        this.id = id;
    }

    @Column(name = "name")
    public String getName () {
        return name;
    }

    public void setName (String name) {
        this.name = name;
    }

    @Column(name = "sport")

    @Column(name ="id")
    public String getId () {
        return id;
    }

    public void setId (String id) {
        this.id = id;
    }

    @Column(name ="name")
    public String getName () {
        return name;
    }

    public void setName (String name) {
        this.name = name;
    }

    @Column(name ="sport")

    @Column(name = "id")
    public String getId () {
        return id;
    }

    public void setId (String id) {
        this.id = id;
    }

    @Column(name = "name")
    public String getName () {
        return name;
    }

    public void setName (String name) {
        this.name = name;
    }

    @Column(name = "sport")

    @Column(name ="id")
    public String getId () {
        return id;
    }

    public void setId (String id) {
        this.id = id;
    }

    @Column(name ="name")
    public String getName () {
        return name;
    }

    public void setName (String name) {
        this.name = name;
    }

    @Column(name ="sport")

    @Column(name = "id")
    public String getId () {
        return id;
    }

    public void setId (String id) {
        this.id = id;
    }

    @Column(name = "name")
    public String getName () {
        return name;
    }

    public void setName (String name) {
        this.name = name;
    }

    @Column(name = "sport")

    @Column(name ="id")
    public String getId () {
        return id;
    }

    public void setId (String id) {
        this.id = id;
    }

    @Column(name ="name")
    public String getName () {
        return name;
    }

    public void setName (String name) {
        this.name = name;
    }

    @Column(name ="sport")

    @Column(name = "id")
    public String getId () {
        return id;
    }

    public void setId (String id) {
        this.id = id;
    }

    @Column(name = "name")
    public String getName () {
        return name;
    }

    public void setName (String name) {
        this.name = name;
    }

    @Column(name = "sport")

    @Column(name ="id")
    public String getId () {
        return id;
    }

    public void setId (String id) {
        this.id = id;
    }

    @Column(name ="name")
    public String getName () {
        return name;
    }

    public void setName (String name) {
        this.name = name;
    }

    @Column(name ="sport")

    @Column(name = "id")
    public String getId () {
        return id;
    }

    public void setId (String id) {
        this.id = id;
    }

    @Column(name = "name")
    public String getName () {
        return name;
    }

    public void setName (String name) {
        this.name = name;
    }

    @Column(name = "sport")

    @Column(name ="id")
    public String getId () {
        return id;
    }

    public void setId (String id) {
        this.id = id;
    }

    @Column(name ="name")
    public String getName () {
        return name;
    }

    public void setName (String name) {
        this.name = name;
    }

    @Column(name ="sport")

    @Column(name = "id")
    public String getId () {
        return id;
    }

    public void setId (String id) {
        this.id = id;
    }

    @Column(name = "name")
    public String getName () {
        return name;
    }

    public void setName (String name) {
        this.name = name;
    }

    @Column(name = "sport")

.contentType
public String getSport() {
    return sport;
}

public void setSport(String sport) {
    this.sport = sport;
}

@OneToMany
public List<Team> getTeams() {
    return teams;
}

public void setTeams(List<Team> teams) {
    this.teams = teams;
}

protected String id;

protected String name;

protected String sport;

protected List<Team> teams;
package org.Request.session;
import java.util.List;
import javax.ejb.Remote;
@Remote
public interface Request {
    public void createPlayer(Player player);
    public void addPlayer(String playerId, String teamId);
    public Player findPlayer(String id);
    public Team findTeam(String id);
    public Set<Player> findAllPlayers();
    public Set<Player> findByPosition(String position);
}
package org.Request.session;
import java.util.List;
import javax.ejb.Remote;
@Remote
public class RequestBean implements Request{
    private EntityManager em;
    public void createPlayer(Player player) {
        em.persist(player);
    }
    public void addPlayer(String playerId, String teamId) {
        Team team = findTeam(teamId);
        Player player = findPlayer(playerId);
        team.addPlayer(player);
        player.addTeam(team);
    }
    public Player findPlayer(String id) {
        return (Player) em.createNamedQuery("findPlayer").getResultList();
    }
    public Team findTeam(String id) {
        return (Team) em.createNamedQuery("findTeam").getResultList();
    }
    public Set<Player> findAllPlayers() {
        return (Set<Player>) em.createNamedQuery("findAllPlayers").getResultList();
    }
    public Set<Player> findByPosition(String position) {
        return (Set<Player>) em.createNamedQuery("findByPosition").getResultList();
    }
}
B  ZOOM Model for Hospital Information Management System

ZOOM model for HIMS referenced in Chapter 7

```java
namespace hospital;

@Security(Authentication="Yes", RoleManagement="Yes")
@table
struct Person {
    @id
    int id;
    String name;
    String address;
}

@table
struct Patient extends Person{
    int id;
    String insurance;
}

@table
struct Doctor extends Person{
    int id;
    String expertise;
    String academicDegree;
}

@table
struct Nurse extends Person{
    int id;
    String skill;
}

@table
struct Staff extends Person{
    int id;
    String position;
}

@table
struct Appointment {
    int id;
    String date;
    String time;
}

@Association(multiplicityLeft=Multiplicity.One, multiplicityRight=Multiplicity.One, roleLeft = patient, roleRight = appointment)
Relation<Patient, Appointment> patientAppointment;
```
@Association(multiplicityLeft=Multiplicity.One, multiplicityRight=Multiplicity.One,
roleLeft = doctor, roleRight = appointment)
Relation<Doctor, Appointment> doctorappointment;

@table
struct Operation {
  int id;
  String type;
}

@Association(multiplicityLeft=Multiplicity.One, multiplicityRight=Multiplicity.One,
roleLeft = patient, roleRight = operation)
Relation<Patient, Operation> patientoperation;

@Association(multiplicityLeft=Multiplicity.Many, multiplicityRight=Multiplicity.One,
roleLeft = doctor, roleRight = operation)
Relation<Doctor, Operation> doctoroperation;

@Association(multiplicityLeft=Multiplicity.Many, multiplicityRight=Multiplicity.One,
roleLeft = nurse, roleRight = operation)
Relation<Nurse, Operation> nurseoperation;

@table
struct MedicalRecord {
  @id
  int id;
  String desc;
}

@table
struct LabReport extends MedicalRecord{
  int id;
  String insurance;
}

@table
struct MedicalHistory extends MedicalRecord{
  int id;
  String expertise;
  String academicDegree;
}

@table
struct ClinicRecord extends MedicalRecord{
  int id;
  String skill;
}

@Association(multiplicityLeft=Multiplicity.One, multiplicityRight=Multiplicity.One,
roleLeft = staff, roleRight = clinicreport)
Relation<Staff, ClinicRecord> staffclinicreport;

@Association(multiplicityLeft=Multiplicity.One, multiplicityRight=Multiplicity.One,
roleLeft = operation, roleRight = clinicreport)
Relation<Operation, ClinicRecord> operationclinicreport;

@table
struct MedicalDataTemplate {
    int id;
    String desc;
    String SecurityLevel;
    String format;
    String attachment;
}

@Association(multiplicityLeft=Multiplicity.Many, multiplicityRight=Multiplicity.One,
roleLeft = medicalDatatemplate, roleRight = medicalrecord)
Relation<MedicalDataTemplate, MedicalRecord> templaterecord;

@table
struct Bill {
    @id
    int id;
    float amount;
}

@Association(multiplicityLeft=Multiplicity.One, multiplicityRight=Multiplicity.One,
roleLeft = patient, roleRight = bill)
Relation<Patient, Bill> patientbill;

@table
struct Lab {
    @id
    int id;
    String name;
}

@Association(multiplicityLeft=Multiplicity.One, multiplicityRight=Multiplicity.One,
roleLeft = patient, roleRight = lab)
Relation<Patient, Lab> patientlab;

@Association(multiplicityLeft=Multiplicity.One, multiplicityRight=Multiplicity.One,
roleLeft = lab, roleRight = labreport)
Relation<Lab, LabReport> lablabreport;
C Screenshots of Hospital Information Management System

Available Function

Patient Registration

- Add a new patient
- View patient record
- Update patient record

Personnel Management

- Doctor Demographic
- Staff Demographic
- Nurse Demographic

Appointment and Scheduling

- Make an appointment
- View appointment record
- Modify appointment

Medical Report

- Add Medical Record
- View Medical record
- Modify Medical record

Figure 18: HIMS Main Page ASP.NET version
Available Function

Patient Registration

- Add a new patient
- View patient record
- Update patient record

Personnel Management

- Doctor Demographic
- Staff Demographic
- Nurse Demographic

Appointment and Scheduling

- Make an appointment
- View appointment record
- Modify appointment

Medical Report

- Add Medical Record
- View Medical record
- Modify Medical record

Figure 19: HIMS Main Page JSP version
Figure 20: HIMS View Medical Data ASP.NET version

<table>
<thead>
<tr>
<th>Patient ID</th>
<th>Description</th>
<th>Security Level</th>
<th>Format</th>
<th>Attachment</th>
<th>Link to Attachment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Doctor note</td>
<td>Medium</td>
<td>TXT</td>
<td>medicalnote1.txt</td>
<td>Click Here</td>
</tr>
<tr>
<td>2</td>
<td>Doctor note</td>
<td>High</td>
<td>Doc</td>
<td>medicaldoc_01.doc</td>
<td>Click Here</td>
</tr>
<tr>
<td>3</td>
<td>X-Ray</td>
<td>High</td>
<td>JPG</td>
<td>X-Ray_01.jpg</td>
<td>Click Here</td>
</tr>
<tr>
<td>2</td>
<td>Doctor Note</td>
<td>Medium</td>
<td>TXT</td>
<td>medicalnote2.txt</td>
<td>Click Here</td>
</tr>
<tr>
<td>2</td>
<td>X-Ray</td>
<td>Medium</td>
<td>JPG</td>
<td>X-Ray_02.jpg</td>
<td>Click Here</td>
</tr>
</tbody>
</table>
Show all the available medical data

<table>
<thead>
<tr>
<th>id</th>
<th>patientid</th>
<th>desc</th>
<th>securitylevel</th>
<th>format</th>
<th>attachment</th>
<th>thumbnail</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Doctor note</td>
<td>Medium</td>
<td>TXT</td>
<td>medicalnote1.txt</td>
<td>null</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Doctor note</td>
<td>High</td>
<td>Doc</td>
<td>medicaldoc_01.doc</td>
<td>null</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>X-Ray</td>
<td>High</td>
<td>JPG</td>
<td>X-Ray_01.jpg</td>
<td>null</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>Doctor Note</td>
<td>Medium</td>
<td>TXT</td>
<td>medicalnote2.txt</td>
<td>null</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>X-Ray</td>
<td>Medium</td>
<td>JPG</td>
<td>X-Ray_02.jpg</td>
<td>null</td>
</tr>
</tbody>
</table>

Figure 21: HIMS View Medical Data JSP version