Model Driven Development for Enterprise Applications

Asif Dogar

A Thesis
in
The Department
of
Computer Science
and
Software Engineering

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

October 2007

Your file Votre référence
ISBN: 978-0-494-40938-1
Our file Notre référence
ISBN: 978-0-494-40938-1

ABSTRACT

Model Driven Development for Enterprise Applications

Asif Dogar

Model-Driven Development (MDD) is an approach to software development that advocates the use of models as the primary artifacts that drive the development process. Automation and abstraction are the two main pillars of MDD. Automation is mainly achieved in the form of code generation. Automated transformations are used to generate code from models. In this thesis, we establish that efficient code generation is a viable option and that it enhances productivity. Significant amount of code can be auto-generated if software patterns are automated using model transformations. We demonstrate the process of automation of a subset of enterprise-application patterns by making use of a state-of-the-art MDD tool called IBM Rational Software Architect. We illustrate the working and outcome of our automated patterns with the help of a case study: a Team Registration System.

Acknowledgments

I want to extend my gratitude to all people who helped me with my research and the writing of this thesis. First and the foremost I am grateful to my supervisor Dr. Patrice Chalin for his outstanding guidance, support and motivation that he provided to me throughout my Master's research. He has been an inspiration and a role model.

I thank all the members of our Dependable Software Research Group (DSRG), including Stephan Barrett, Perry James, George Karabotsos, Rajiv Abraham, Stuart Thiel and Kianoush Torkzadeh for their help, support and feedback.

I specially want to thank a great researcher and a great friend Daniel Sinnig for his overall support, calculated feedback, exciting ideas, and readiness to help. Without him, my experience in the DSRS lab would not have been the same.

Above all, I am grateful to my family. I thank my parents, my brother and my sisters for their active support, for their belief in my abilities, for their love and for their encouragement.

Table of Contents

List of Figures

List of Acronyms

Acronym	Definition
API	Application Programming Interface
CORBA	Common Object Request Broker Architecture
DM	Data Mapper
DSRG	Dependable Software Research Group
EJB	Enterprise Java Beans
EMF	Eclipse Modeling Framework
JSP	Java Server Pages
JET	Java Emitter Templates
MDD	Model Driven Development
OCL	Object Constraint Language
OMG	Object Management Group
PIM	Platform Independent Model
PoEAA	Patterns of enterprise application architecture
RAS	Reusable Asset Specification
RBML-PI	Role Based Meta-modeling Language-Pattern Instantiator
RSA	Rational Software Architect
RTE	Round Trip Engineering
SQL	Structured Query Language
TRS	Team Registration System
TDG	Table Data Gateway
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSD	XML Schema Definition
UML	Unified Modeling Language

1 Introduction

1.1 Model-Driven Development

Model-driven development (MDD) is an approach to developing software in which models are the primary artifacts that drive the development. As MDD practitioners, we first create a platform-independent model of the system-under-development (using the Unified Modeling Language or a domain-specific modeling language) before applying automated transformations to this model (the source model) to obtain another model (the target model) which is either an enhancement, a specialization, or a refinement of the platform-independent model in a modeling language, or code in a programming language such as Java or C++. In the context of this thesis, we use the term platform independent model (PIM) to refer to a UML model that captures the subject matter such as banking, telephony, etc and is free from any platform-specific details.

Two key themes are at the heart of MDD: raising the level of abstraction and raising the level of automation [Selic 2006]. **Abstraction** is the removal of irrelevant details. "A good abstraction is one that emphasizes details that are significant to the reader or user and suppresses details that are, at least for the moment, immaterial or diversionary" [Shaw 1984]. **Automation** is the most effective technological means for enhancing productivity and reliability [Selic 2003]. Automation also provides a shield against the tedious and the mundane.

High-level languages provide a shield against the very low-level details such as the realization of function calls and memory management. Escaping from such details was the motive for moving from low-level assembly languages to high-level languages such

as Java and C#. Still, the level of abstraction provided by these high-level languages is not enough. Real word concepts are very far removed from the statements and data structures that constitute the core of these programming languages. The detailed nature of programming languages impedes abstraction [Selic 2006]. MDD raises the level at which systems are designed and built. Using it, developers can focus on understanding the core of the system rather than worrying about how the solution to be devised.

In MDD, automation is mainly achieved in the form of **Code Generation** by way of a code generation tool. Most state-of-the-art MDD tools have code generators embedded in them. Generation of code from a model is called Forward Engineering and generation of a model from code is called Reverse Engineering. A combination of forward and reverse engineering is called Round-Trip-Engineering (RTE).

Abstraction and automation together bring us closer to achieving the very significant objective of reuse. New technologies are introduced frequently, and there is a demand by customers to use them. As a result, existing software is often ported either to a new technology or to a newer version of the existing technology [Kleppe *et al.* 2003]. However, the core of the software, the business logic, does not change much despite changes in the requirements. This core of the software can be reused. A model that is closer to the problem domain and is free from any implementation-specific details makes this sort of reuse possible. Automated code generators can transform the same model into code suitable for many different target platforms. In a nutshell, the two pillars of MDD—abstraction and automation—support reuse, and reuse increases productivity.

1.2 The Problem of skeletal code generation

Automated code generation is a key feature of MDD. Automated code generation enhances productivity, but if the code generation is "skeletal," which means that all a code-generation tool generates is empty class and method bodies, then this will not make a significant difference in productivity. As MDD practitioners, we desire much more than the generation of method and class skeletons (which is what most MDD tools offer). In order to attain the maximum benefits of MDD, we need to fully exploit its potential for automation by auto-generating as much code as possible. Auto-generating maximum possible code will speed up the development and eliminate the effort involved by automating the realization of common patterns [Selic 2006].

1.3 Proposed Solution: Automation of Software Patterns

Software patterns are a means to achieve reuse. The idea behind patterns is to provide well-defined solutions to recurring problems. Patterns are formalized for two important reasons. The first reason is to capture design expertise. The solutions that have proven to work are formalized in the form of patterns. The second reason is to save the time and effort that may go into solving a similar problem more than once. Although finer details of the problem that a pattern solves may vary, the core is always the same. Despite the core being the same, code for patterns has to be rewritten from scratch every time. MDD advocates the automation of such mechanical tasks.

We believe that the problem of skeletal code generation can be overcome, at least in part, by auto-generating code for software patterns. Since a pattern encompasses the knowledge of the problem at hand, and its solution, much more code can be auto-

generated than mere empty method bodies. Besides, automated generation of pattern code will also solve the problem of manual pattern application.

1.4 Objectives and Contributions

The main objective of this thesis is to demonstrate that the problem of skeletal code generation can be overcome in part through automation of software patterns. We establish that the automated realization of software patterns in code can be achieved by creating model transformations using an MDD tool such as IBM Rational Software Architect (RSA).

In order to achieve our objectives, we select a subset of enterprise-application patterns called *Patterns of Enterprise Application Architecture* (PoEAA) [Fowler 2002] and automate them using RSA. We validate the usefulness of our automated patterns by using them in the development of a case study, a Team Registration System.

The key contributions of this thesis are, the elaboration of

- A UML profile for PoEAA.
- A set of automated transformations to generate code for a subset of PoEAA patterns.
- A case study to validate our claim that automation of software patterns can improve productivity.

1.5 Thesis Organization

The organization of the thesis is as follows. In chapter 2, "Background and Related work," we explain model transformations in detail. We also define with examples two categories of model transformations. In the end, we highlight work in the field of pattern

automation. Chapter 3, "Writing Custom Model Transformations," highlights the MDD capabilities of IBM Rational Software Architect. In particular, we describe its support for authoring custom UML profiles and model transformations. In chapter 4, "Model Transformations for Enterprise Applications," we first introduce the PoEAA patterns along with a summary of the subset of patterns we automate. We also explain the process of automating the chosen patterns using Rational Software Architect. In chapter 5, "Case Study: Team Registration," we make use of our automated patterns in the development of a Team Registration System and define the applicability of the defined transformations. In chapter 6, "Conclusion and Future Work," we conclude the thesis and predict future avenues of research.

2 Background and Related Work

2.1 Model Transformations

In the previous chapter, we mentioned the two pillars of MDD, abstraction and automation. These two pillars together help us achieve improved productivity by speeding-up the development process and by enabling reuse.

"Abstraction is one of the fundamental ways that we as humans cope with complexity" [Booch 2005]. Removal of irrelevant details helps us manage complexity. In MDD, we achieve a higher level of abstraction by separating application business logic from the underlying platform technology. A platform-independent model is used to capture domain knowledge, and any platform-specific details are left out of the model. This has many benefits, such as

- The model can be used and reused for realization of the system on any platform.
- The analysis, time, and effort that went into building the model is not wasted when the software has to be ported to a newer version of the current technology or to a different technology.
- Developers can focus on the core of the system, the business logic and create extensible, flexible designs without getting distracted by technological details.

All these benefits can be achieved through the use of a platform-independent model. The model may be free of any platform-specific details, but the question remains, where does the information about platform-specific details that must be part of the implementation reside? A model transformation includes this platform-specific information.

"A **transformation** is the automatic generation of a target model from a source model, according to a transformation definition. A **transformation definition** is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A **transformation rule** is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language" [Kleppe *et al.* 2003]. Model transformations take one or more source models as input, generally in UML and produce one or more target models as output [Sendall & Kozakczynski 2003].

Model transformations can be classified into two types, **Model-to-Model** and **Model-to-Code** transformations [Czarnecki & Helsen 2003]. Model-to-Model transformations translate between models that are instances of the same or different meta-model. A meta-model defines the structure, semantics and constraints for a family of models [Mellor & Scott 2004]. Model-to-Code transformations can be seen as a specialization of the Model-to-Model transformations since code itself is also a model albeit at a much lower level of abstraction. However, Model-to-Model transformations are out of the scope of this thesis since our focus is on code generation by way of Model-to-Code transformations. Model-to-Code generation takes a platform-independent model as input and generates code for a particular technology platform such as Java or .Net as output. Figure 1 illustrates the

application of a model-to-code transformation to a UML model, resulting in the generation of code in the target programming language.

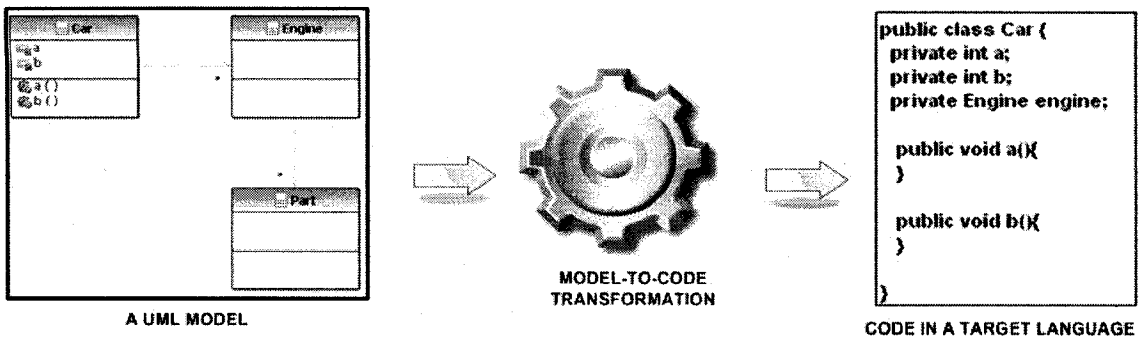

Figure 1: Model-to-Code transformation

Code generation can increase productivity significantly. The ideal scenario would be to have most, if not all of the code generated from the model. Unfortunately, we have to accept that a standard "one serves all" transformation is infeasible [Weis *et al* 2003]. The more generic a transformation is, the less code it will generate; and the more specialized a transformation is, the more code it will generate. A very generic model transformation leads to skeletal code generation. An example of such transformation is the "UML-to-Java" transformation that is part of IBM Rational Software Architect's (RSA) transformation library.

Model transformations can be most useful when they are built exclusively for, or customized for a particular application area such as enterprise applications or telecommunications. Systems that are written for an application area have parts that are common. Code for these parts is usually written from scratch for each application.

Specialized transformations can be used to auto-generate this common code to maximize productivity.

Besides specialization, a transformation has to incorporate developers' common practices. Developers will not throw away their common practices simply because a transformation restricts their use. They would rather discard the tool [Weis *et al* 2003]. It is therefore necessary that tools let their users to not only apply predefined transformations, but also define new transformations and customize the existing ones.

2.2 Types of Model-to-Code Transformation Approaches

There are two types of approaches for model-to-code transformations: Visitor-Based approaches and Template-Based approaches [Czarnecki & Helsen 2003].

2.2.1 Visitor-Based Approaches

Visitor-based approaches use a mechanism to traverse the internal representation of the model and write code to a text output stream. An example of such an approach is Jamda [http://jamda.sourceforge.net].

Jamda is an open-source framework for building model transformations that generate Java code from UML models. Jamda transformations are written in Java, which provides a Java API. A transformation built in Jamda takes a UML model in XMI (XML metadata interchange) format. XMI is the OMG's [OMG 2007] XML-based standard for interchange of UML models. Some modeling tools support the export and import of UML models in XMI format [Sendall & Kozakczynski 2003].

2.2.2 Template-Based Approaches

Most of the tools use a template-based approach to implement model-to-code transformations. "A template usually consists of the target text containing splices of meta-code to access information from the source and to perform code selection and iterative expansion" [Czarnecki & Helsen 2003]. Java Emitter Templates (JET) is a template-based approach to code generation. In this thesis, we have used the template-based approach. RSA has built-in support for JET.

2.3 Other Related Work

Most of the patterns that we automate in this thesis deal with persistence and object/relational mapping. Relational databases are a popular choice for storing data [Pete & Heudecker 2006]. The model used by relational databases, called the "relational model" stores data in two-dimensional tables. Data in these tables can be linked via primary and foreign keys. A primary key uniquely identifies a given row in a table whereas a foreign key refers to a primary key stored in another table. An object model, on the other hand, is meant to model the problem domain more faithfully. There will be inheritance and composition dependencies between objects. There are one-to-one, one-to-many, and many-to-many associations between objects. This radical difference between the relational model and the object model requires some kind of object-relational (O/R) mapping code. Such code either needs to be written "by hand" or tools such as Hibernate [Hibernate 2007] need to be used.

Hibernate is an object/relational mapping framework for Java applications. It stores objects in the database freeing the developer from writing and maintaining code that

stores and retrieves objects. Hibernate bridges the gap between the relational and the object schemas. With respect to O/R mappings, Hibernate supports a range of object-oriented features such as inheritance, custom object types, and collections. It also provides support for creation of object proxies. In the final chapter of this thesis, we compare and contrast Hibernate with our automated O/R mapping patterns.

The focus of this thesis is on code generation. For developers interested in code generation, there is an online network called "The Code Generation Network" [www.codegenerationnetwork.com]. It lists a number of resource, tools and books that focus on code generation.

[Dae-Kyoo & Whittle 2005] created a prototype tool called Role-Based Metamodeling Language-Pattern Instantiator (RBML-PI). This tool generates domain-specific design pattern elements from a pattern specification described in the Role-Based Metamodeling Language, a UML-based pattern-specification language.

[Weis *et al* 2003] described a scheme for transforming models using a graphical modeling language called Kafka that veils explicit reference to the metamodels involved. They also created a modeling tool called Kase that makes use of Kafka.

3 Writing Custom Model Transformations

In this chapter, we present the Model-Driven Development (MDD) capabilities of IBM Rational Software Architect (RSA). We first provide a brief introduction to the tool, followed by a detailed description of its basic MDD capabilities as well as its support for domain specific modeling. The latter is achieved through UML profile authoring which enables the creation of custom patterns and transformations.

3.1 Introduction to IBM Rational Software Architect

IBM Rational Software Architect (RSA) is a state-of-the-art tool supporting MDD. It is built on top of the well-known open-source Eclipse platform. RSA is a member of the new generation of modeling and model driven development tools which was launched by IBM in 2004 to follow IBM's successful Rational Rose application series. RSA offers the ability to create, test, package, deploy and apply patterns, profiles and transformations. As an MDD tool, RSA provides features such as

- Modeling using the Unified Modeling Language (UML 2.0)
- A Patterns framework
- Model transformations and code generation
- Reuse through asset based development using Reusable Asset Specifications (RAS) [RAS 2007]
- Support for UML 2.0 profiles
- Extensibility framework for authoring UML profiles containing custom patterns and transformations

In the upcoming sections, we will look at some of these capabilities in detail.

3.2 Basic Support for MDD

3.2.1 Modeling Using UML 2.0

Software development in Eclipse[1] is achieved through Eclipse perspectives. A perspective is preconfigured to offer access to a closely related set of features. Examples of commonly used perspectives include the Java perspective (for Java code development), the Debugging perspective and the CVS project synchronization perspective. RSA[2] enhances Eclipse with a modeling perspective through which developers can create and manipulate model projects—i.e., Eclipse projects specialized to contain models.

To capture and communicate the different aspects of an application and to cater for the needs of various stakeholders, various types of UML 2.0 diagrams can be created to illustrate a model. Specifically, the types of diagrams supported in RSA are:

- Class diagram.
- Composite structure diagram
- Component diagram
- Deployment diagram
- Object diagram
- Use case diagram
- Activity diagram

---

[1] Eclipse is a Java-based, extensible open source development platform. It is a framework and a set of services for building a development environment from plug-in components.
[2] IBM Rational Software Architect (RSA) is an integrated design and development tool that leverages MDD with UML for creating well-architected applications and services. It is built on top of the Eclipse platform.

- State machine diagram
- Sequence diagram
- Communication diagram

A screenshot of a sample development session which is making use of RSA's modeling perspective is given in Figure 2.

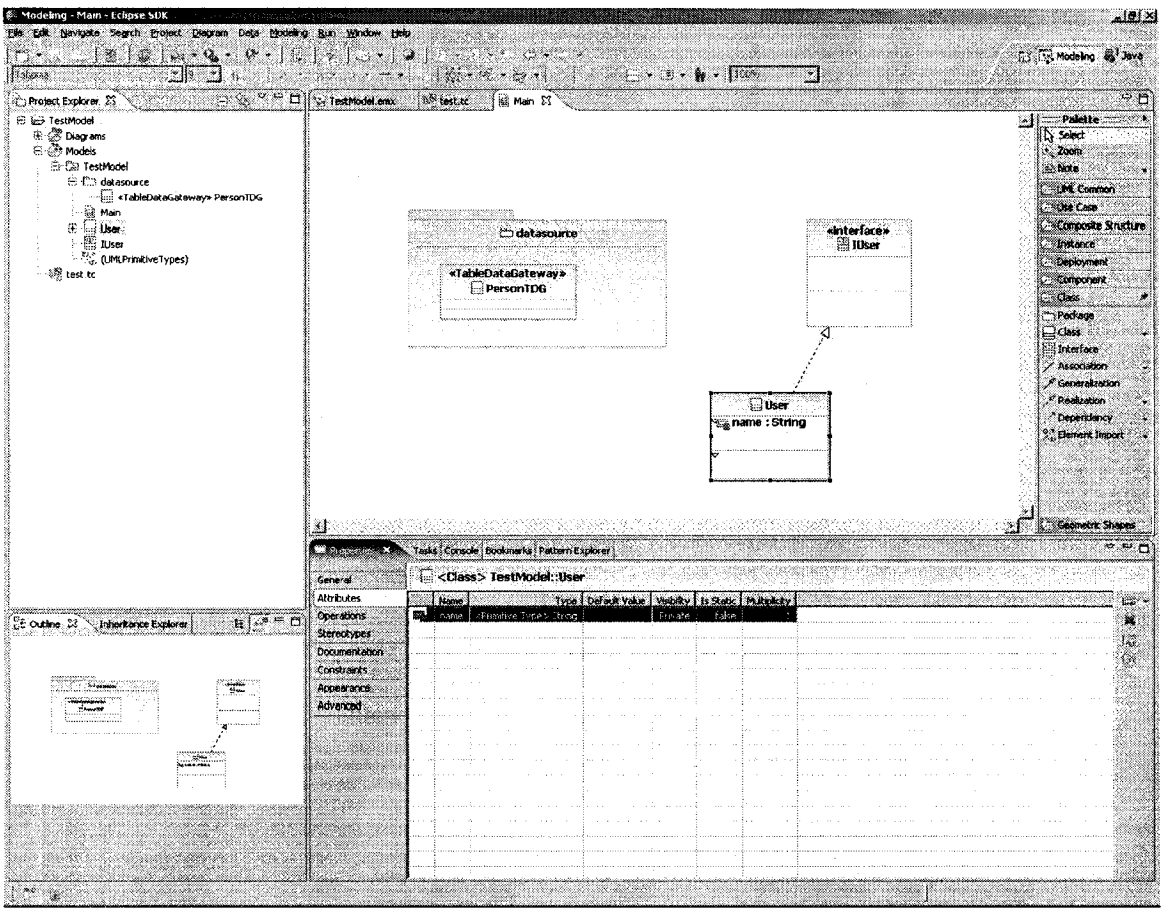

Figure 2: An active development session using RSA's modeling perspective

3.2.2 Patterns Framework

Patterns provide solutions to recurring problems in a particular context. The patterns framework of RSA is used for:

- **Applying patterns to UML model elements.** A Pattern Explorer provides access to RSA's pattern library, which includes, among others, the Gang of Four [Gamma *et al.* 1995] design patterns. Figure 3 shows the Pattern Explorer in RSA. Figure 4 shows the application of Singleton pattern to a UML class. We drag and drop a pattern instance on the diagram editor. Next, UML elements are supplied to the pattern parameters as arguments. In this particular example, the Singleton pattern has one parameter called "Singleton" and the acceptable type is UML Class. We drag and drop the ToyFactory UML class into the pattern instance. As a result, Singleton pattern is applied to ToyFactory class (in this case, the stereotype <<singleton>> is added to the class).
- **Authoring custom patterns and transformations.** Details of pattern authoring are presented in the next section. Once authored, custom patterns become a part of RSA's pattern library.
- **Packaging and publishing patterns for reuse.** A custom pattern can be packaged and deployed as a plug-in, which conforms to the OMG's Reusable Asset Specification (RAS) [RAS 2007]. Once packaged, the reusable asset can be exchanged as a file or can be published to a shared asset repository for other people to search for and import into their RSA workbench.

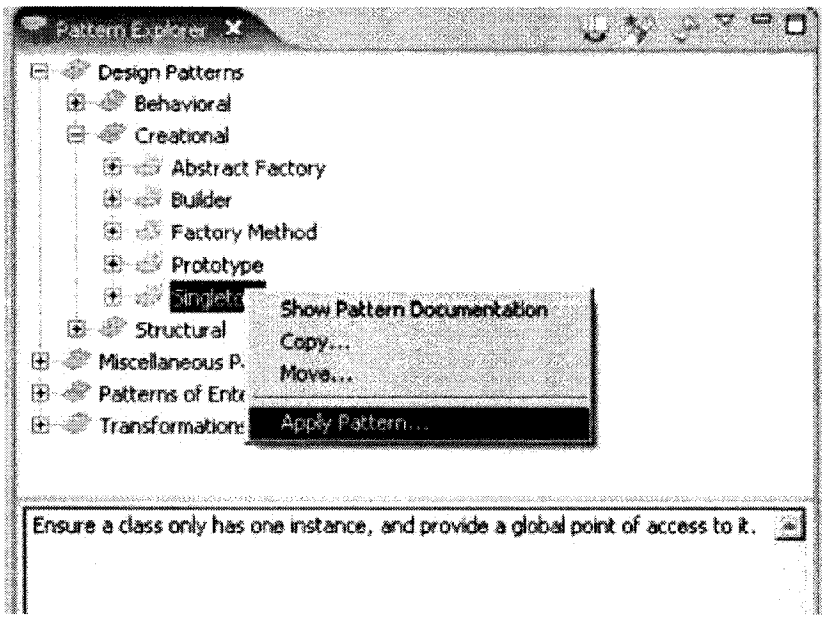

Figure 3: Pattern Explorer in RSA

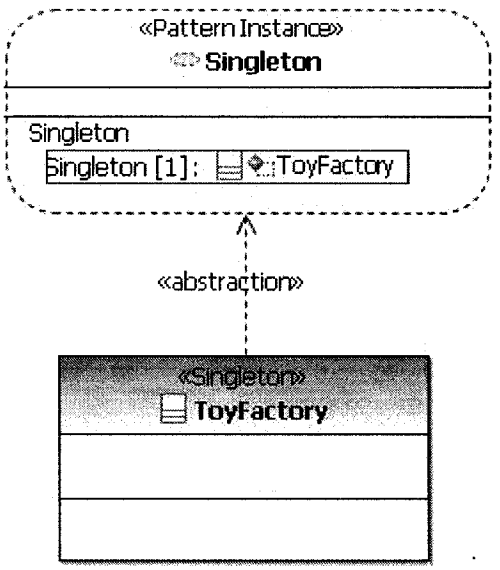

Figure 4: Singleton pattern from RSA's pattern library applied to a UML class

3.2.3 Model Transformations and Code Generation

As of version 7, RSA has built-in transformations for UML to Java 1.4, Java 5, EJB, CORBA, XSD and C++. These transformations can be applied to UML elements at different levels in the model hierarchy such as the entire model or selected packages, classes, and/or interfaces. RSA supports Round Trip Engineering (RTE) by providing the reverse transformation "Java to UML".

3.3 Support for Custom Profile Creation

"UML profiles tailor the language to specific areas, some for business modeling, others for particular technologies" [OMG 2007]. A UML profile extends UML with a set of coherent group of stereotypes for a particular purpose, such as business modeling [Fowler 2003]. A stereotype is a new kind of UML model element that is invented by the modeler and is based on an existing kind of model element [Rumbaugh *et al* 2005].

An RSA profile is a collection of stereotypes, enumerations and classes (Figure 5).

The typical steps of creating a custom UML profile in RSA are,

1. Creation of a custom UML profile using the "New Profile" wizard
2. Adding stereotypes the profile and specifying constraints for the custom profile using the "Properties" view.

Once a UML profile is created, the next step is to add stereotypes. For each stereotype that is added, RSA's Properties view can used to

- Specify the type of UML element(s) that a stereotype extends

- Add one or more attributes to stereotypes. Attributes that belong to a stereotype will be added to the list of UML properties of the element to which the stereotype is applied.
- Specify an icon or a graphic to identify each stereotype that we create.
- Specify constraints for the stereotype using the Object Constraint Language (OCL) [OCL 2007].
- Specify Keywords for the stereotype

Enumerations are used to specify the type of stereotype attributes. A stereotype attribute's value whose type is an enumeration would be one of the enumeration literals owned by that enumeration. The profile-owned enumerations are not available outside the profile. Similarly, classes can be added to a profile and can be used to specify the type of stereotype attributes. The profile-owned classes are not available outside of the profile

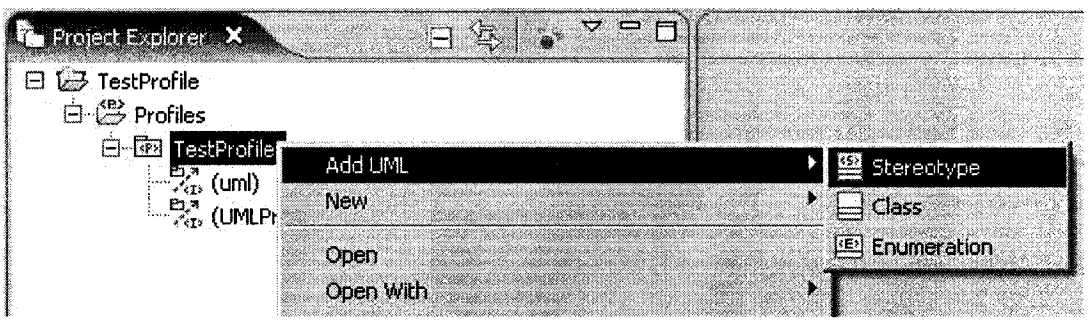

Figure 5: Adding elements to a custom UML profile

An RSA profile can be applied to a UML model in two ways, either through the RSA user interface or programmatically. Figure 6 shows the "Applied Profiles" list in the "Properties" view for a UML model. A new profile can be applied to the model by

selecting it from the list of existing UML profiles. Stereotypes can be applied to the UML model elements by making use of the "Properties" view for the element. Figure 7 shows the Properties view for a UML class. The "Apply Stereotypes" button will pop up a list of all the available stereotypes to choose from. Profiles and stereotypes can also be applied programmatically as a part of the pattern authoring process.

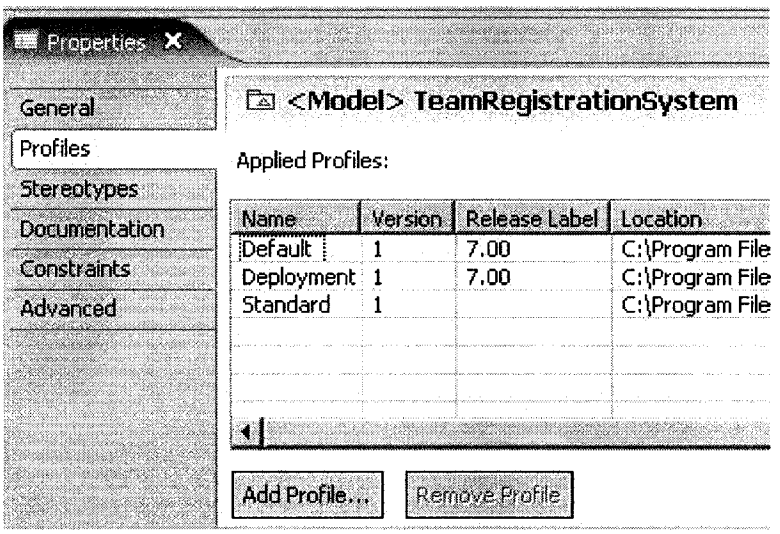

Figure 6: List of Profiles applied to the model shown in the Properties view

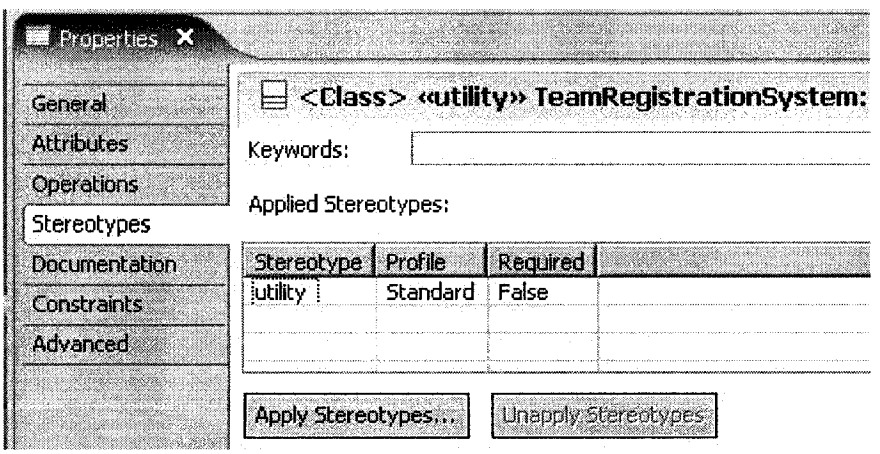

Figure 7: Applied stereotypes for a UML elements shown in the Properties view

In the next chapter, we present the creation of a custom profile, which we called PoEAA (short for Patterns of Enterprise Application Architecture) to support modeling and code generation for Enterprise Applications. We will also demonstrate the programmatic application of RSA profiles and stereotypes to UML elements.

3.4 Custom Pattern Authoring

Custom patterns are deployed as RSA plug-ins. Hence, to create custom patterns, we first create a "Plug-in with Patterns" project in the RSA workspace. Next, we use the "Pattern Authoring" view to add parameters to the pattern. Each pattern parameter defines the targeted UML element's type and the acceptable multiplicity values. We can also define dependencies between pattern parameters.

After the new pattern has been added to the project and parameters have been added to the pattern, a Java-based pattern framework is auto-created in the plug-in project. This framework provides functions for the pattern author to use in defining the pattern's behavior. This basic framework is called the implementation model of the pattern.

The implementation model includes the pattern library class that instantiates pattern classes. For each pattern, it includes a Java class representing the pattern. Inside the Java class there is a nested Java class for each pattern parameter and each dependency between the parameters. Each nested class that represents a pattern parameter contains two empty `expand()` methods and each nested class that represents a dependency between patterns contains three empty `update()` methods. These `expand()` and `update()` methods are

known as the "hot spots". Pattern authors add pattern behavior in the form of code in these hot spots. Figure 8 shows the empty `expand()` methods

```
 * @generated
 */
public boolean expand(PatternParameterValue value) {
    //TODO : implement the parameter's expand method
    return true;
}

 * @generated
 */
public boolean expand(PatternParameterValue.Removed value) {
    //TODO : implement the parameter's expand method
    return true;
}
```

Figure 8: The "hot spots"

In the hot spots, the pattern behavior is implemented by making use of the UML2 API [http://www.eclipse.org/uml2] and RSA patterns framework API. Figure 9 shows sample code in a hot spot method. It can be seen that a pattern author makes use of the mentioned API's to implement the behavior of a pattern programmatically.

```
public boolean expand (PatternParameterValue value) {

// Cast the pattern parameter to a UML class
   org.eclipse.uml2.uml.Class targetClass =
(org.eclipse.uml2.uml.Class)value.getValue();

// create a UML operation in the class

targetClass.createOwnedOperation("OperationName",listOfParameterNames,
listOfParameterTypes););

// create a new UML package in the owner model of the pattern parameter

targetClass.getModel().createNestedPackage("application");
    return true;
  }
```

Figure 9: Sample code for in a hot spot method

Once the pattern implementation is completed, patterns are packaged and deployed as RSA plug-ins. After deployment, the authored RSA patterns become a part of the RSA pattern collection..

RSA patterns do not generate code. RSA patterns affect the UML model only. Code generation is achieved through RSA model-to-code transformations. Besides providing the basic UML-to-Java transformations, RSA provides a transformation framework to allow for creation of custom transformations.

3.5 Custom Transformation Authoring

As mentioned earlier, RSA provides many transformations like the UML-to-Java transformation. However, these transformations offer only the basic functions to generate the corresponding code artifacts from the models. For example, a UML-to-Java transformation when applied to a UML class will only generate Java method skeletons for UML operations.

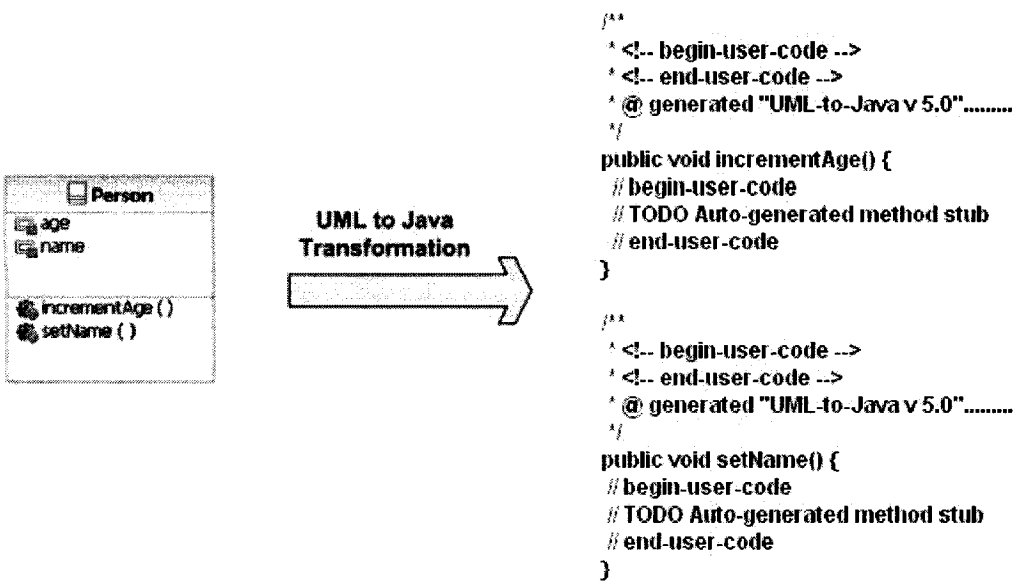

Figure 10: Basic UML to Java transformation in RSA

The code that is generated is very limited as can be seen in Figure 10. As a MDD practitioner, much more is desired. A developer would like to auto-generate as much code as possible rather than generating only the method skeletons. Rational Software Architect provides a framework for extending existing transformations to add additional behavior and for creating new transformations.

The transformation framework extensibility is based on the Eclipse plug-in architecture. Thus, the custom transformation are created and deployed as RSA plug-ins. In order to extend or create transformations, new extensions to the points declared by the transformation core are defined.

In order to extend an existing transformation, an extension point is added to extend the core RSA transformation providers framework. Then rules are defined to specify the names of the classes that will provide the implementation for the transformation

extension. Transformation rules are used to associate code-generation classes with UML elements. The transformation rules are called each time a specified UML element is found in the model and rules invoke the code to generate the desired code depending on the element type.

Authoring custom transformations is similar to extending existing transformations but requires more effort and understanding of the transformation extensibility framework on the part of the transformation author.

In order to create a custom transformation, a new plug- in project is created. Once the new plug-in project wizard is completed, the core classes and methods acting as placeholders for extensibility code are created. To understand the implementation of these classes, it is important to have some familiarity with the transformation API.

There are two important packages in the transformation API,

- com.ibm.xtools.transform.core
- com.ibm.xtools.uml.transform.core

Package com.ibm.xtools.transform.core

The important classes in this package are shown in Figure 11.

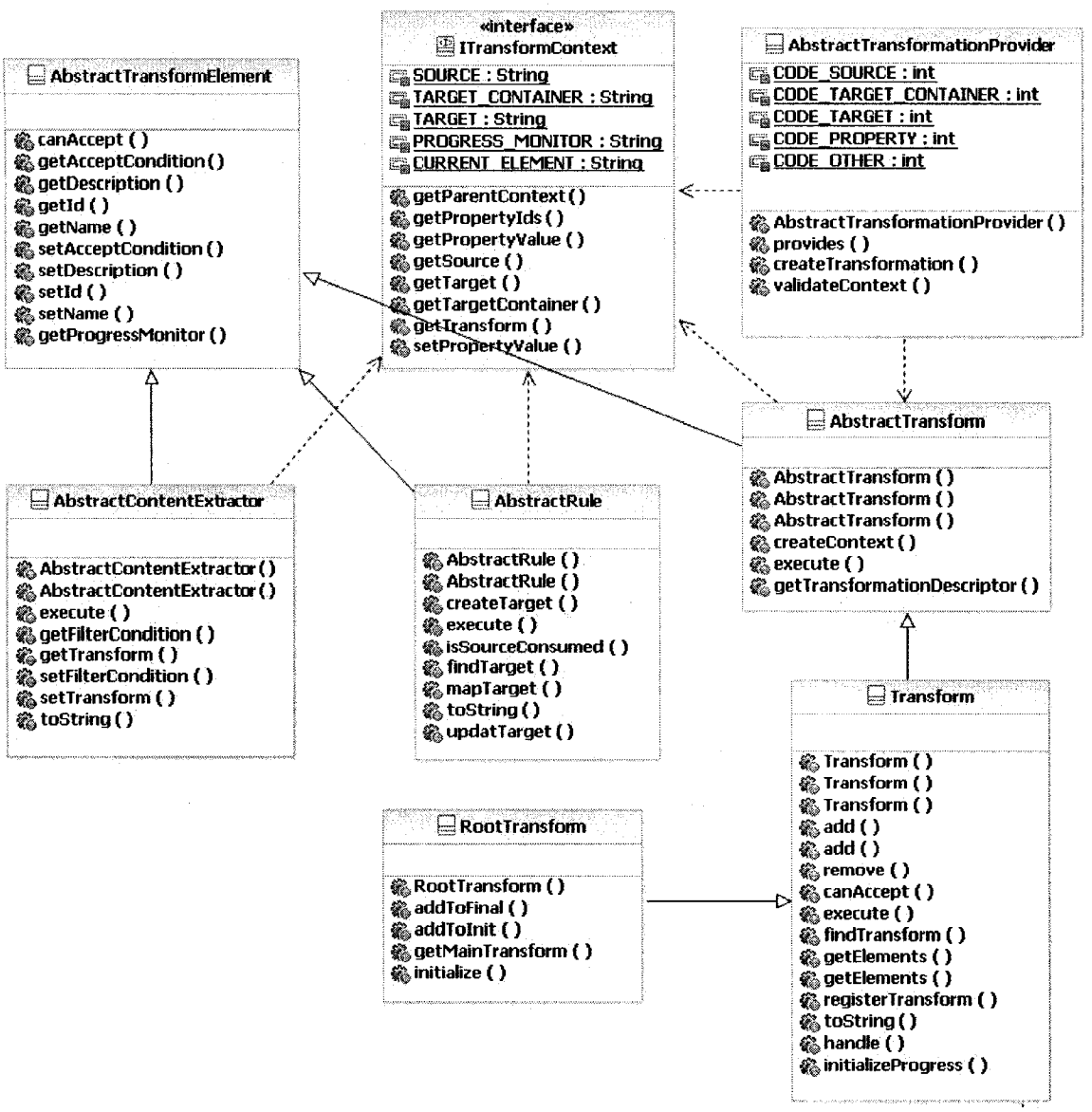

Figure 11: com.ibm.xtools.transform.core

AbstractTransform

This is the base class for all the transformation implementations with one abstract method:

```
public void execute(ITransformContext context)
```

ITransformContext

`ITransformContext` is responsible for maintaining the execution's contextual information like the current source object and target objects. For example, while creating a UML-to-Java transformation, the `ITransformContext` object will contain the source UML model (or part of the model) to which the transformation is applied and the target Java project. In the transformation implementation code, the source-model information is retrieved from the `ITransformContext`.

AbstractContextExtractor

An extractor extracts source model elements from the context for processing. For example, if the source object is a UML class, the extractor will return all contained attributes and operations.

Transform

Class `Transform` extends `AbstractTransform` and implements the `execute()` method by iterating over a collection of transforms, rules and extractors and recursively calling their `execute()` methods.

AbstractRule

A rule is an instance of a sub class of `AbstractRule`. Rules provide most of the implementation for custom transformations. Rules modify a target object given a source

object. The rules contain the code to create the transformation target artifacts such as Java packages and classes as well as the code to generate code or other text.

Package com.ibm.xtools.uml.transform.core

This package contains classes exclusively for writing transformations for UML 2.0 models. The important classes in this package are shown in Figure 12.

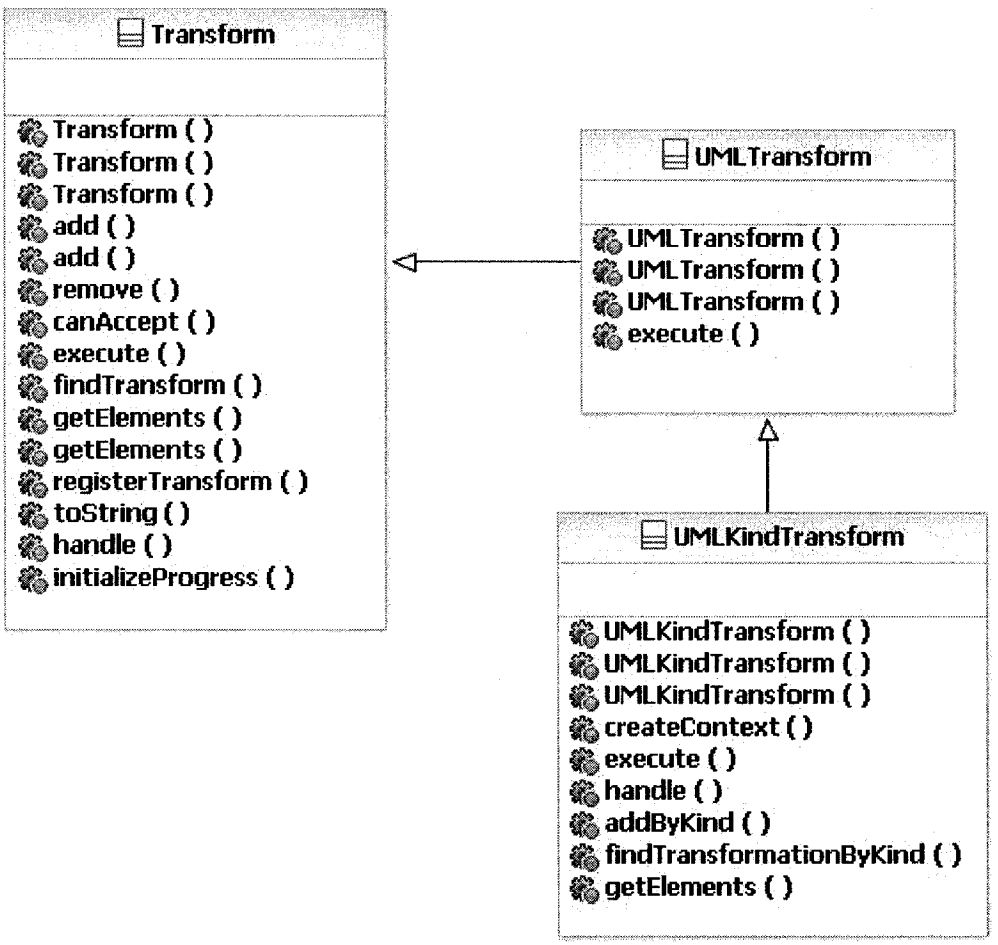

Figure 12: com.ibm.xtools.uml.transform.core

UMLKindTransform

This class traverses a source UML model or a part of the model and for each source element it executes all the rules, transforms and extractors registered for that type of model element. It contains rules, transforms and extractors associated with different UML elements.

Three important classes form the core of a custom transformation plug-in project.

Plug-in Activator class

This is a singleton that is instantiated when the plug-in is activated.

Transformation Provider class

The transformation provider class lists all the transformations. This class instantiates a transformation and also has methods to validate the source and the target objects in the context.

Root Transform class

The Root Transform class is a utility class that transformation authors can use as the root of their transformation. This transform has a predefined structure with three phases, initialization, execution and finalization. The initialization and finalization phases are optional. The execution phase is required to execute the main transform. Multiple rules or child transforms may be defined in the initialization or the final phases.

4 Model Transformations for Enterprise Applications

In the previous chapter, we presented the Model-Driven Development (MDD) capabilities of IBM Rational Software Architect (RSA). In this chapter, we make use of those capabilities to create a UML profile, author custom patterns and write custom transformations to generate pattern code. We first provide a brief introduction to the Patterns of Enterprise Application Architecture by Martin Fowler [Fowler 2002], and a summary of the authored patterns.

4.1 Introduction to Patterns of Enterprise Application Architecture

Enterprise applications generally involve large amounts of data, are accessed by many users concurrently, have many user interface screens, integrate with other enterprise applications and have complex business logic [Fowler 2002]. Because of these characteristics, developing enterprise software can be a challenging task.

Martin Fowler, based on his years of experience with enterprise software development, formalized a number of patterns to address various recurring problems [Fowler 2002]. His patterns address issues such as, the organization of domain logic, mapping to relational databases, web presentation and concurrency management. We will use the abbreviation PoEAA (Patterns of Enterprise Application Architecture) to refer to these patterns.

Before introducing individual patterns, we must present the layering scheme proposed by Martin Fowler that conforms to the standard three-layer scheme for enterprise

applications. Each of the patterns presented later can belong to one or more of the three principal layers. The three layers in the layering scheme are

- **Presentation** layer is used to display information and handle user requests (such as HTTP requests, mouse clicks, keyboard hits etc). It is responsible for providing the interface for supporting the interaction between the user and the application.
- **Domain Logic** layer is also referred to as the business logic layer. As the name suggests, this layer holds the domain objects and the helper objects that assist the domain objects fulfill the responsibilities. This layer is responsible for manipulating domain objects, performing calculations on them, etc.
- **Data Source** layer is responsible for communicating with the databases, messaging systems, transaction managers, other applications, etc

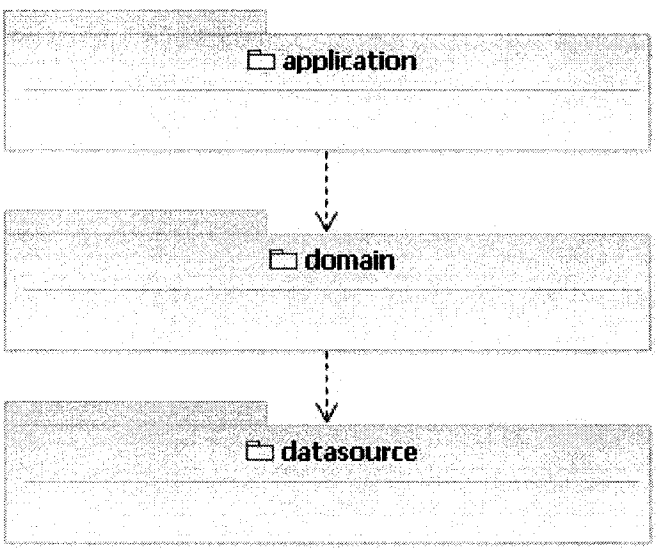

Figure 13:Standard layering scheme for Enterprise Applications

4.1.1 Summary of Selected Enterprise Patterns

Martin Fowler has proposed 51 enterprise patterns [Fowler 2002]. In this section, we present an overview of the patterns that we selected for the purpose of automation. The

criterion for selection of these patterns was the frequency of their usage in the Dependable Software Research Group (DSRG) lab at Concordia University.

In order to explain the patterns that we selected for the purpose of automation, we first need to explain the *Domain Model* pattern. Martin Fowler has proposed the use of a *Domain Model* to organize the domain logic of complex applications. Using a *Domain Model* means segregating in a distinct layer (shown in Figure 13) the objects that model the domain for which the application is being built. A *Domain Model* is useful for reducing domain complexity because it uses object orientation for describing the design model as close to the chosen abstractions of the domain as possible [Nilsson 2006].

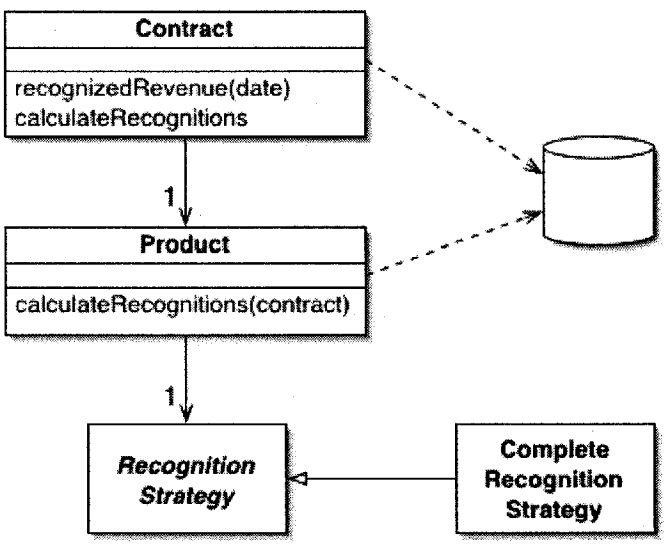

Figure 14: Domain Model Pattern [Fowler 2002]

A rich *Domain Model* is used for complex business logic. Hence, it will lead to a web of interconnected objects with many dependencies, strategies, inheritance and patterns. Mapping such a rich *Domain Model* to the database is difficult since the mapping is not

one-to-one between domain model classes and database tables. A set of *Data Mappers* can be used to map a rich domain models to the database.

4.1.1.1 Data Mapper (DM) Pattern

A *Data Mapper (DM)* acts as an intermediary between a domain-model object and the database by moving data between the two. The object model and the database model use two different schemas for structuring data. A *Data Mapper* maps the two schemas to each other. It also isolates the domain-model objects from the database. *Domain Model* objects are completely oblivious of the database and even the *Data Mapper* itself.

Figure 15 shows the typical methods in a *Data Mapper*: `insert()`, `update()` and `delete()`. All these methods accept a domain object as an argument, extract data from it and then either invoke commands on the database directly or call operations on an object such as a *Table Data Gateway*, which is responsible for database access, and pass the values to it.

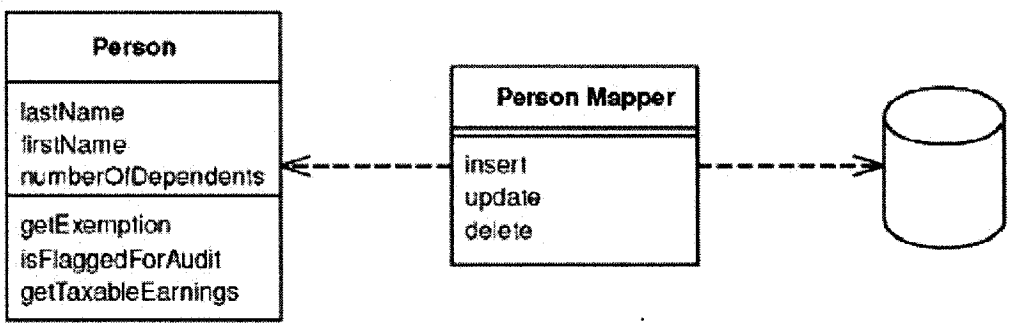

Figure 15: An example *Data Mapper*, an intermediary between a domain model class and the tadatabase[Fowler 2002]

4.1.1.2 Table Data Gateway (TDG) Pattern

A *Table Data Gateway* (TDG) mediates all access to a database table and prevents direct coupling of the database with the rest of the system. There is generally one TDG per table

in the database and all the interaction with a table is done through its TDG. Another benefit of using a TDG is that all the SQL (Structured Query Language) queries for a database table are confined in a single class.

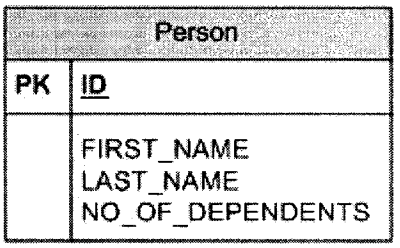

Figure 16: An example database table for which a TDG is needed

A typical TDG contains methods that hold SQL statements to perform selects, inserts, updates and deletes on a database table. Each of these methods plugs its input parameters into an SQL statement and then executes this statement against a database connection. Figure 17 shows an example TDG for the table shown in Figure 16.

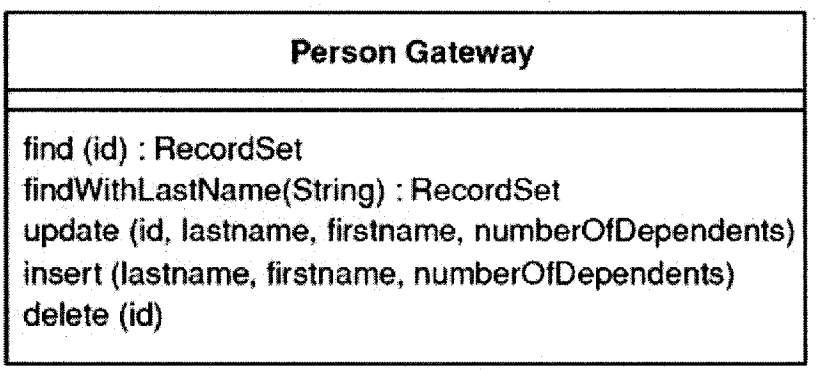

Figure 17: An example Table Data Gateway [Fowler 2002]

4.1.1.3 Lazy Load with Virtual Proxy

While loading data for an object from the database into memory, it is useful to have related objects also loaded in the memory. However, with such a design, loading one object may have the effect of loading a large number of related objects when only a few objects are actually needed. This can have a negative impact on the performance.

Lazy Load is a pattern to deal with this problem. With *Lazy Load*, data is loaded into memory only when it is needed. There are four ways to implement a lazy load: lazy initialization, virtual proxy, value holder and ghost. Martin Fowler recommends the use of *Lazy Load with a Virtual Proxy* if a Data Mapper is being used.

A *Virtual Proxy* is an object that has the same interface as the real object that it is standing in for, except that it does not contain any real data but it has the knowledge of how to get the real object. Only when one of its methods is called, the real object is loaded into the memory.

4.1.1.4 Finder

A *Table Data Gateway* is used to perform insert, update and delete operations on a database, it may also have find methods to find domain objects in the database. However, several find methods can clutter the *Table Data Gateway*. It is recommended put all the find methods in a separate Finder class.

4.2 Automation of PoEAA Using Rational Software Architect

PoEAA patterns are used extensively in our DSRG lab. In an effort to avoid writing repetitive code, Dr. Patrice Chalin and Stuart Thiel created a framework called SoenEA. Generic and repetitive code for these patterns was refactored and packaged into this framework. Since then, the *Data Mappers*, the *Table Data Gateways* and other pattern classes extend from classes in the SoenEA framework.

Even though the use of SoenEA has helped increase productivity, there is still a lot of similar code that has to be rewritten each time a PoEAA pattern is used. In what follows, we demonstrate how patterns can be automated using model transformations. The pattern code that is generated by the transformations will still make use of and be compatible with the SoenEA framework.

The use of the SoenEA framework provided a base for making some simplifying assumptions to help the process of pattern automation. In the upcoming sections, besides describing the automation of each pattern, we will clearly state the simplifying assumptions for each pattern.

4.2.1 Overview of the Automation Process

In order to extend the UML meta-model to enable modeling of the enterprise application patterns, we first create a custom UML profile called PoEAA. Next, we specify stereotypes, constraints and keywords in the profile. After the profile has been deployed, we make use of Rational Software Architect's patterns framework to author a subset of the PoEAA patterns. For each enterprise pattern, we repeat a two-step process.

- **Create a RSA pattern** to insert pattern-specific information into the UML model.

- **Create a RSA transformation** that outputs auto-generated code by taking pattern-specific model information as input.

RSA patterns only add information to the model. They do not generate code. Code generation is achieved through a Model-to-Code transformation. Hence, as a first step of the pattern automation process, we will create a RSA pattern and implement pattern behavior programmatically. We then create a RSA transformation that takes the output of the pattern as input and generates code. Figure 18 shows the entire process of code generation and the artifacts that are created during step 1 and step 2 of the pattern automation process.

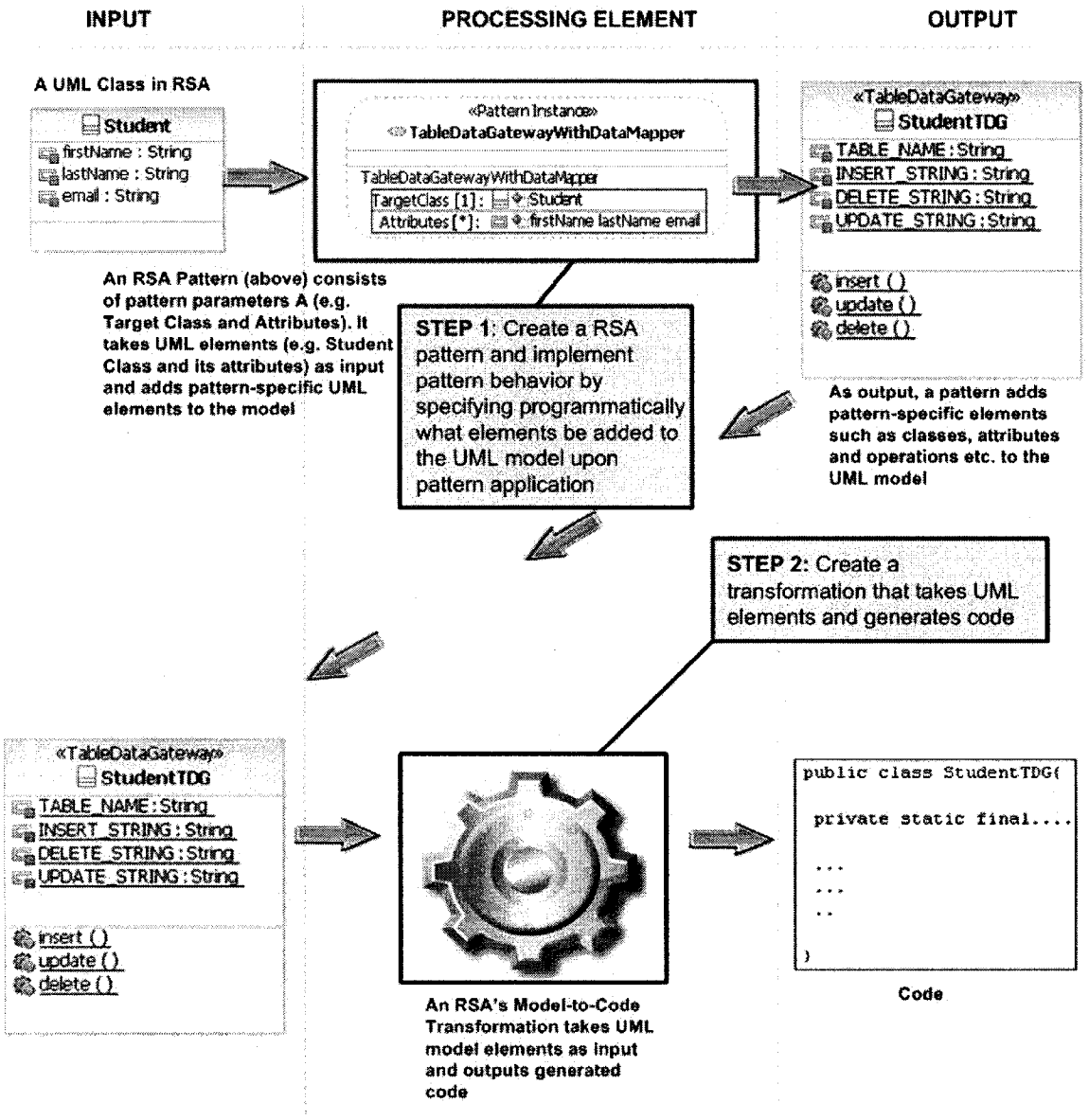

Figure 18: Process of code generation for patterns in RSA and the two steps of pattern automation

4.2.2 Setup for Pattern Automation

In RSA, created patterns and transformations are deployed as RSA plug-ins. By using the "Plug-in development" perspective, we create two separate projects. A "Plug-in with Patterns" project called *PatternsOfEnterpriseApplicationArchitecture* for authoring patterns, and a "Plug-in with Transformations" project called *PoEAATrans-*

formationPlugin for creating transformation that will generated code for the authored patterns.

4.2.3 Creation of PoEAA UML Profile

In RSA, the process of profile creation is very straightforward. We first create a UML profile called *PoEAA* inside the *PoEAATransformationPlugin* project. Next, we add one stereotype for each PoEAA pattern we want to automate. After adding the stereotypes to the profile, we specify stereotype details using RSA's "Properties" view. Figure 19 shows the created PoEAA profile.

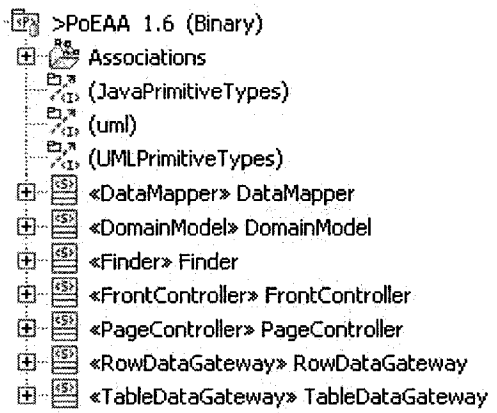

Figure 19: PoEAA profile

Stereotype Name	Visibility	Attributes			Is Static	Is Abstract	Is Leaf	Keyword
		Meta-Class	Required	Multi-plicity				
TableDataGateway	Public	Class	False	1	False	False	False	TableDataGateway
DataMapper	Public	Class	False	1	False	False	False	DataMapper
VirtualProxy	Public	Class	False	1	False	False	False	VirtualProxy
Finder	Public	Class	False	1	False	False	False	Finder

Table 1: Properties of stereotypes added to PoEAA profile

4.3 Transformation 1: Table Data Gateway with Data Mapper

A recommended approach is to use a *Table Data Gateway* (TDG) with a *Data Mapper* (DM). A DMuses the services of a TDG to read from and write to a database. Because of this strong dependency, we decided to tie these two patterns into a single RSA pattern called *TableDataGatewayWithDataMapper.*

4.3.1 Simplifying Assumptions

Based of the use of SoenEA framework, we make the following assumptions,

- Each *Domain Model* class extends the Java class `DomainObject` that is part of the SoenEA framework. Thus, each domain class contains two attributes, "version" and

"id" of Java primitive type "long". SoenEA makes use of these two attributes to store and retrieve a domain object from the database.

- In order to achieve a 100 % auto-generated *Data Mapper* and *Table Data Gateway*, we assume that the there is a one-to-one relationship between a domain object and the associated database table i.e. one domain objects is mapped to one database table.
- We assume that all the attributes of a class are of primitive types such as String, Integer or Boolean. If an object contains a reference to another object then 100 % code is not generated since the multiplicity is not one-to-one and in such case, not only the objects but also the mapping between objects has to be persisted also.

4.3.2 Implementing the Pattern

We first use the "Pattern Authoring" view in RSA to create the pattern in the *PatternsOfEnterpriseApplicationArchitecture* project. In RSA, a pattern is defined with the help of pattern parameters. Pattern parameters accept as arguments UML elements that are participants in a pattern. For instance, an RSA pattern called *GetterSetter* that creates a getter and a setter method for an attribute of a UML class may have two pattern parameters: one that accepts a UML class in which the getter and setter methods will be created and another that accepts one or more UML attributes of the UML class for which getter and setter methods are needed.

We define two pattern parameters for our *TableDataGatewayWithDataMapper* pattern. One is the `Targetclass` that accepts a UML class and the other is `Attributes` that accepts one or more UML attributes. As mentioned earlier, only UML attributes that have a multiplicity equal to one and are of primitive types can be passed to `Attributes` as arguments because otherwise the mapping becomes complex and the current version of

our design is unable to deal with this complexity. A developer who wants to make use of the *TableDataGatewayWithDataMapper* pattern will have to pass a *Domain Model* UML class as the `Targetclass` parameter and will also pass one or more UML attributes of the same class as the `Attributes` parameter as arguments.

In the previous chapter, we mentioned the pattern expansion code briefly. The pattern expansion code is used to define the behavior of a pattern. For each pattern parameter, RSA's pattern framework auto-generates methods that are referred to as the "hot spots." The patterns framework calls these methods when a pattern is applied.

We make use of the IBM Rational Patterns Java API to define pattern behavior in the hot spots. It provides the functionality to add, remove and change UML model elements in a UML model. For our *TableDataGatewayWithDataMapper* pattern, we use this API to cause programmatically the following changes to the UML model when the pattern is applied to a UML class. Figure 20 shows the pattern implementation code the hot-spot methods. Notice that the code makes use of the supplied Java API to manipulate the UML model elements.

- Create a UML package called `datasource` if it does not already exist in the model.
- If the package exists, check if it already contains a *Table Data Gateway* for the class that the pattern is being applied. If it does, update it, and if it does not then create a UML class with the `<<TableDataGateway>>` stereotype, and all the required attributes and operations inside the package.
- Similarly, check inside the `domain` package if a *Data Mapper* for the class that the pattern is applied to already exists. If so, update it; else create a UML class with a stereotype `<<DataMapper>>`, and all the required methods.

Figure 21 shows the output of the *TableDataGatewayWithDataMapper* pattern when applied to a UML class called Student. When the pattern is applied to the *Student* class, two UML classes *StudentMapper* and *StudentTDG* with all the required attributes and operations are created in the model. Note that operations in both UML classes make use of the "id" and "version" attributes which are assumed to be in the each domain model class (one of the simplifying assumption).

```
public boolean expand(PatternParameterValue value) {

//get the pattern parameter argument and apply cast

Class domainClass = (org.eclipse.uml2.uml.Class) value.getValue();
org.eclipse.uml2.uml.Model model = domainClass.getModel();

//load the Java primitive types library and get the long Java type
Model javaLibrary = (Model) load(URI
.createURI(org.eclipse.uml2.uml.resource.UMLResource.JAVA_PRIMITIVE_TYPES_LIB
ARY_URI));

PrimitiveType longPrimitiveType = (PrimitiveType)
javaLibrary.getOwnedType("long");

// if a package with name datasource does not exist in the model create it

org.eclipse.uml2.uml.Package datasourcePackage;

if(model.getMember("datasource")== null){
datasourcePackage = model.createNestedPackage("datasource");
}else{
datasourcePackage = model.getNestedPackage("datasource");
}

// if the TDG class does not exist already , create it

if(datasourcePackage.getMember(domainClass.getName()+"TDG")== null){
tdgClass =
datasourcePackage.createOwnedClass(domainClass.getName()+"TDG",false);
}else{
tdgClass.getMember(domainClass.getName()+"TDG");
}


// Apply the PoEAA profile to the model

if (ProfileResource != null) {
List contents = ProfileResource.getContents();
if (contents.size() == 1 && contents.get(0) instanceof Profile) {
poeaaProfile = (Profile) contents.get(0);
}
}

model.applyProfile(poeaaProfile);

tdgClass.applyStereotype(poeaaProfile.getOwnedStereotype("TableDataGateway",
true ,
false));
```

```
// Code for creating a Mapper class


Class mapperClass =
domainClass.getPackage().createOwnedClass(domainClass.getName()+"Mapper",
false);

EList ownedParameterNames = new  BasicEList();
EList ownedParameterTypes = new  BasicEList();

ownedParameterNames.add(domainClass.getName());
ownedParameterTypes.add(domainClass);

Operation insertOperation = mapperClass.createOwnedOperation("insert",
ownedParameterNames, ownedParameterTypes);
insertOperation.setVisibility(VisibilityKind.PUBLIC_LITERAL);
insertOperation.setIsStatic(true);

Operation updateOperation = mapperClass.createOwnedOperation("update",
ownedParameterNames, ownedParameterTypes);
updateOperation.setVisibility(VisibilityKind.PUBLIC_LITERAL);
updateOperation.setIsStatic(true);

Operation deleteOperation = mapperClass.createOwnedOperation("delete",
ownedParameterNames, ownedParameterTypes);
deleteOperation.setVisibility(VisibilityKind.PUBLIC_LITERAL);
deleteOperation.setIsStatic(true);


ownedParameterNames.clear();
ownedParameterTypes.clear();

ownedParameterNames.add("id");
ownedParameterTypes.add(longPrimitiveType);

Operation mapOperation =
mapperClass.createOwnedOperation("map",ownedParameterNames
,ownedParameterTypes,domainClass);
mapOperation.setVisibility(VisibilityKind.PUBLIC_LITERAL);
mapOperation.setIsStatic(true);


Operation getAllOperation =
mapperClass.createOwnedOperation("get"+domainClass.getName()+"s", null,
null,domainClass);
getAllOperation.setVisibility(VisibilityKind.PUBLIC_LITERAL);
getAllOperation.setIsStatic(true);
getAllOperation.getReturnResult().setIsOrdered(true);
getAllOperation.getReturnResult().setIsUnique(false);
getAllOperation.getReturnResult().setUpper(LiteralUnlimitedNatural.UNLIMITED)
getAllOperation.getReturnResult().setTemplateParameter(domainClass.getTemplat
Parameter());



Operation mapKnownOperation =
mapperClass.createOwnedOperation("get"+domainClass.getName()+"s",
ownedParameterNames, ownedParameterTypes ,domainClass);
mapKnownOperation.setName("mapKnown");
mapKnownOperation.setVisibility(VisibilityKind.PUBLIC_LITERAL);
mapKnownOperation.setIsStatic(true);




mapperClass.applyStereotype(poeaaProfile.getOwnedStereotype("DataMapper", tru
, false));

return true;
}
public boolean update(PatternParameterValue value,
```

```
PatternParameterValue dependencyValue) {
Property attribute = (Property) value.getValue();
Class incomingClass = (Class) dependencyValue.getValue();
Operation insertOperation = null;
Operation deleteOperation = null;
Operation updateOperation = null;

Model javaLibrary = (Model) load(URI
.createURI(org.eclipse.uml2.uml.resource.UMLResource.JAVA_PRIMITIVE_TYPES_LI
ARY_URI));


PrimitiveType longPrimitiveType = (PrimitiveType)
javaLibrary.getOwnedType("long");

EList nameListForInsert;
EList typeListForInsert;

EList nameListForUpdate;
EList typeListForUpdate;

EList nameListForDelete;
EList typeListForDelete;


if(tdgClass.getMember("insert")== null){
nameListForInsert = new BasicEList(targetClass.namesList);
typeListForInsert = new BasicEList(targetClass.typeList);
nameListForInsert.add(0,"id");
typeListForInsert.add(0,longPrimitiveType);
insertOperation = tdgClass.createOwnedOperation("insert", nameListForInsert,
typeListForInsert);
insertOperation.setIsStatic(true);

} else {

tdgClass.getMember("insert").destroy();
nameListForInsert = new BasicEList(targetClass.namesList);
typeListForInsert = new BasicEList(targetClass.typeList);
nameListForInsert.add(0,"id");
typeListForInsert.add(0,longPrimitiveType);
insertOperation = tdgClass.createOwnedOperation("insert", nameListForInsert,
typeListForInsert);
insertOperation.setIsStatic(true);
}

if(tdgClass.getMember("update")== null){

nameListForUpdate = new BasicEList(targetClass.namesList);
typeListForUpdate = new BasicEList(targetClass.typeList);
nameListForUpdate.add("0,id");
typeListForUpdate.add(0,longPrimitiveType);
nameListForUpdate.add(1,"version");
typeListForUpdate.add(1,longPrimitiveType);
updateOperation = tdgClass.createOwnedOperation("update",nameListForUpdate ,
typeListForUpdate );
updateOperation.setIsStatic(true);

} else {
tdgClass.getMember("update").destroy();
nameListForUpdate = new BasicEList(targetClass.namesList);
typeListForUpdate = new BasicEList(targetClass.typeList);
nameListForUpdate.add(0,"id");
typeListForUpdate.add(0,longPrimitiveType);
nameListForUpdate.add(1,"version");
typeListForUpdate.add(1,longPrimitiveType);
updateOperation = tdgClass.createOwnedOperation("update",nameListForUpdate ,
typeListForUpdate );
updateOperation.setIsStatic(true);
}

if(tdgClass.getMember("delete")== null){
```

```
nameListForDelete = new BasicEList();
typeListForDelete = new BasicEList();

nameListForDelete.add("version");
typeListForDelete.add(longPrimitiveType);

nameListForDelete.add("id");
typeListForDelete.add(longPrimitiveType);

deleteOperation= tdgClass.createOwnedOperation("delete", nameListForDelete,
typeListForDelete);
deleteOperation.setIsStatic(true);
} else {
tdgClass.getMember("delete").destroy();
nameListForDelete = new BasicEList();
typeListForDelete = new BasicEList();

nameListForDelete.add("id");
typeListForDelete.add(longPrimitiveType);

nameListForDelete.add("version");
typeListForDelete.add(longPrimitiveType);


deleteOperation= tdgClass.createOwnedOperation("delete",nameListForDelete,
typeListForDelete);
deleteOperation.setIsStatic(true);
}


Model umlLibrary = (Model) load(URI
.createURI(org.eclipse.uml2.uml.resource.UMLResource.UML_PRIMITIVE_TYPES_LIBI
RY_URI));

PrimitiveType stringPrimitiveType = (PrimitiveType)
umlLibrary.getOwnedType("String");


Property deleteString;

Property updateString;

if(tdgClass.getMember("TABLE_NAME")== null){
Property tableName= tdgClass.createOwnedAttribute("TABLE_NAME",
stringPrimitiveType);
tableName.setIsStatic(true);
tableName.setIsLeaf(true);
tableName.setVisibility(org.eclipse.uml2.uml.VisibilityKind.PRIVATE_LITERAL);
tableName.setStringDefaultValue("DbRegistry.getTablePrefix()+"+incomingClass.
etName());
} else {
tdgClass.getMember("TABLE_NAME").destroy();
Property tableName= tdgClass.createOwnedAttribute("TABLE_NAME",
stringPrimitiveType);
tableName.setIsStatic(true);
tableName.setIsLeaf(true);
tableName.setStringDefaultValue("DbRegistry.getTablePrefix()+\""+incomingClas
.getName()+"\"");
tableName.setVisibility(org.eclipse.uml2.uml.VisibilityKind.PRIVATE_LITERAL);
}


EList attributeListForStrings;


if(tdgClass.getMember("INSERT_STRING")== null){
attributeListForStrings = new BasicEList(targetClass.attributeList);
Property property1 = UMLFactory.eINSTANCE.createProperty();
Property property2 = UMLFactory.eINSTANCE.createProperty();
property1.setName("id");
property1.setType(longPrimitiveType);
property2.setName("version");
```

```
property2.setType(longPrimitiveType);
attributeListForStrings.add(0,property1);
attributeListForStrings.add(1,property2);
Property insertString = tdgClass.createOwnedAttribute("INSERT_STRING",
stringPrimitiveType);
insertString.setIsStatic(true);
insertString.setIsLeaf(true);
insertString.setVisibility(org.eclipse.uml2.uml.VisibilityKind.PRIVATE_LITERA
);
insertString.setStringDefaultValue(StringUtilityForTDG.getAttibuteListAsStrir
(attributeListForStrings, "INSERT_STRING"));

}else{
tdgClass.getMember("INSERT_STRING").destroy();
attributeListForStrings = new BasicEList(targetClass.attributeList);
Property property1 = UMLFactory.eINSTANCE.createProperty();
Property property2 = UMLFactory.eINSTANCE.createProperty();
property1.setName("id");
property1.setType(longPrimitiveType);
property2.setName("version");
property2.setType(longPrimitiveType);
attributeListForStrings.add(0,property1);
attributeListForStrings.add(1,property2);
Property insertString = tdgClass.createOwnedAttribute("INSERT_STRING",
stringPrimitiveType);
insertString.setIsStatic(true);
insertString.setIsLeaf(true);
insertString.setVisibility(org.eclipse.uml2.uml.VisibilityKind.PRIVATE_LITERA
);
insertString.setStringDefaultValue(StringUtilityForTDG.getAttibuteListAsStrir
(attributeListForStrings, "INSERT_STRING"));

}


if(tdgClass.getMember("DELETE_STRING")== null){
attributeListForStrings = new BasicEList();
Property property1 = UMLFactory.eINSTANCE.createProperty();
Property property2 = UMLFactory.eINSTANCE.createProperty();
property1.setName("id");
property1.setType(longPrimitiveType);
property2.setName("version");
property2.setType(longPrimitiveType);
attributeListForStrings.add(0,property1);
attributeListForStrings.add(1,property2);
deleteString = tdgClass.createOwnedAttribute("DELETE_STRING",
stringPrimitiveType);
deleteString.setIsStatic(true);
deleteString.setIsLeaf(true);
deleteString.setVisibility(org.eclipse.uml2.uml.VisibilityKind.PRIVATE_LITERA
);
deleteString.setStringDefaultValue(StringUtilityForTDG.getAttibuteListAsStrir
(attributeListForStrings, "DELETE_STRING"));

} else {
tdgClass.getMember("DELETE_STRING").destroy();
attributeListForStrings = new BasicEList();
Property property1 = UMLFactory.eINSTANCE.createProperty();
Property property2 = UMLFactory.eINSTANCE.createProperty();
property1.setName("id");
property1.setType(longPrimitiveType);
property2.setName("version");
property2.setType(longPrimitiveType);
attributeListForStrings.add(0,property1);
attributeListForStrings.add(1,property2);
deleteString = tdgClass.createOwnedAttribute("DELETE_STRING",
stringPrimitiveType);
deleteString.setIsStatic(true);
deleteString.setIsLeaf(true);
deleteString.setVisibility(org.eclipse.uml2.uml.VisibilityKind.PRIVATE_LITERA
);
deleteString.setStringDefaultValue(StringUtilityForTDG.getAttibuteListAsStrir
```

```
(attributeListForStrings, "DELETE_STRING"));

}


if(tdgClass.getMember("UPDATE_STRING")== null){
attributeListForStrings = new BasicEList(targetClass.attributeList);
updateString= tdgClass.createOwnedAttribute("UPDATE_STRING",
stringPrimitiveType);
updateString.setIsStatic(true);
updateString.setIsLeaf(true);
updateString.setVisibility(org.eclipse.uml2.uml.VisibilityKind.PRIVATE_LITERA
);
updateString.setStringDefaultValue(StringUtilityForTDG.getAttibuteListAsStrir
(attributeListForStrings, "UPDATE_STRING"));

} else {
tdgClass.getMember("UPDATE_STRING").destroy();
attributeListForStrings = new BasicEList(targetClass.attributeList);
updateString= tdgClass.createOwnedAttribute("UPDATE_STRING",
stringPrimitiveType);
updateString.setIsStatic(true);
updateString.setIsLeaf(true);
updateString.setVisibility(org.eclipse.uml2.uml.VisibilityKind.PRIVATE_LITERA
);
updateString.setStringDefaultValue(StringUtilityForTDG.getAttibuteListAsStrir
(attributeListForStrings, "UPDATE_STRING"));

}
return true;
}
```

Figure 20: Code for a hotspot method in *TableDataGatewayWithDataMapper* implementation

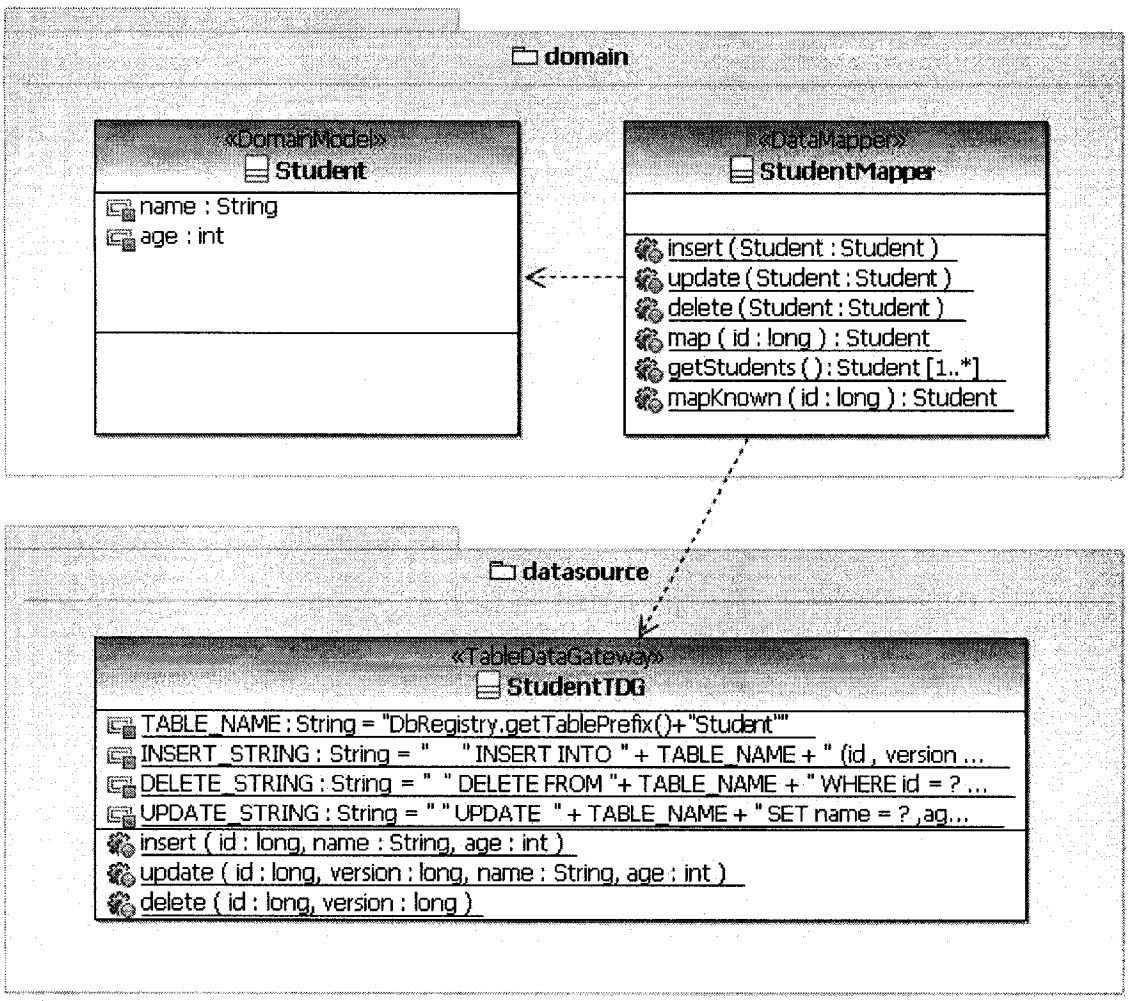

Figure 21: Outcome of the *TableGataGatewayWithDataMapper* pattern, when applied to a UML class called Student

Our next step is to create a model-to-code transformation that takes the generated TDG and DM UML classes as input and generates the required pattern code as output.

4.3.3 Creating the Transformation

In order to create our own UML-to-PoEAA transformation, we make use of the *PoEAA-TransformationPlugin* project. In the transformation provider class we instantiate the `PoEAAtoJavaRootTransform` class that extends `RootTransform`. In the `canAccept()`

method of `PoEAAtoJavaRootTransform`, we add code to declare a UML class, a UML package and a UML model as the acceptable UML types for our transformation. To have a more specialized transformation that accepts specific UML types, we create a `PoEAAtoJavaTransform` class that extends `UMLKindTransform` and register it with the root transform class `PoEAAtoJavaRootTransform`. The `PoEAAtoJavaTransform` traverses a UML model, and for each UML class it invokes one or more transformation rules. As we mentioned in the previous chapter, transformation rules provide most of the implementation for a transformation. For our UML-to-PoEAA transformation, we add three rules to the `PoEAAtoJavaTransform`. All three rules will be invoked for each UML class in a UML model. Our three rules are called Setup Target Rule, Class Rule and Save Output Rule. The order in which these rules are added to the transformation is crucial because for each UML element in the source model, the rules are run in that order. For example if we have a rule to modify a UML element and we have a separate rule to save this modification. We will first add the rule that does the modification and then add the rule that saves this modification. In our case, we add rules to the transformation in that order as we define them here.

4.3.3.1 Setup Target Rule

As the name suggests, this rule contains code to create transformation-target artifacts such as Java classes, interfaces and packages in the workspace. For example, for the StudentMapper UML class, this rule will create a corresponding Java class file called `StudentMapper.java` in the target Java project so that generated code can be written to this file.

4.3.3.2 Class Rule

This rule checks if the incoming class is marked with any of the stereotypes from the PoEAA profile. If so, it invokes a Java Emitter Template (JET) to generate pattern code. Figure 22 shows a part of the code that does the template invocation based on the stereotype.

```
  if (element.getAppliedStereotype("PoEAA::TableDataGateway") != null) {
        textOutput = new
TableDataGatewayTemplate().generate(templateContext);
  }

  else if (element.getAppliedStereotype("PoEAA::DataMapper") != null) {
        textOutput = new DataMapperTemplate().generate(templateContext);
  }

  else if (element.getAppliedStereotype("PoEAA::Finder") != null) {
        textOutput = new FinderTemplate().generate(templateContext);
  }

  else if (element.getAppliedStereotype("PoEAA::LazyLoad") != null) {
        textOutput = new
LazyLoadWithProxyTemplate().generate(templateContext);
  }

  else if (element.getAppliedStereotype("PoEAA::FrontController") != null
{
        textOutput = new
FrontControllerTemplate().generate(templateContext);
  }
```

Figure 22: A part of `ClassRule` code

Java Emitter Templates (JET) is a source-code generation technology that is a part of the Eclipse Modeling Framework (EMF). JET offers a Java Server Pages (JSP)-like syntax to create templates that express the code that we want to generate. Similar to JSP, JET allows the insertion of string values within code using expressions and the use of Java code to perform loops, conditional logic or declare variables using scriptlets. JET files have the `.jet` extension. JET is not limited to generating Java source code only and can be used to generate XML, SQL and text files.

For each UML class with a stereotype that belongs to the PoEAA profile, ClassRule will put all the information of the UML model of the class into a Hashmap and pass it to the

`generate()` method of the JET that corresponds to the stereotype. For example, for a UML class with a `<<StudentMapper>>` stereotype, ClassRule will invoke the `generate()` method of the `DataMapperTemplate` class.

JET makes use of RSA's Java APIs to access the model information. JET retrieves the required information from the model and combines it with some static text to produce the desired code. Figure 23 shows a sample JET code. In this small example, the class and the package names are retrieved from the UML model, and the static text has been enclosed in boxes. The output of sample JET is shown in Figure 24. The complete JET source for this transformation is given in Appendix 8.1

```
<%
// retrieve the context
Map context = (Map)argument;
Class TDGClass =(org.eclipse.uml2.uml.Class)context.get("class");
String className = TDGclass.getName();

Package pkg=(org.eclipse.uml2.uml.Package) TDGclass.getPackage();
String pkgName = pkg.getName();
%>
package <%=pkgName%>;
public class <%=className%> {
}
```

Figure 23: A Sample JET

```
package domain;
public class Student{
}
```

Figure 24: Output of the sample JET

The `generate()` method of the JET returns the generated code as a String. ClassRule saves this output String in the transformation-context object.

4.3.3.3 Save Output Rule

SaveTargetRule retrieves the generated code as a Java String from the context object and makes use of Java's I/O support to write it to a Java file that was created by the Setup Target Rule.

4.4 Transformation 2: Lazy Load with Virtual Proxy

4.4.1 Simplifying Assumptions

In order to automate the lazy load with virtual proxy pattern, we make the following simplifying assumptions,

- The real and the proxy class will both implement a common interface whose name is the name of the real class preceded by "I".
- The common interface will have all the public operations of the real class.
- The proxy class will contain two attributes, "id" of type long and reference to the real object. The name of the reference is "inner" concatenated with name of the real class.
- The proxy class contains one public constructor, which accepts the "id" of the real object as an argument.
- The proxy class contains a private method `getInner()` that returns the reference to the real object.

4.4.2 Implementing the Pattern

In order to automate the Lazy Load with Virtual Proxy pattern, we first create a *LazyLoadWithVirualProxy* RSA pattern in the PatternsOfEnterpriseApplication-

Architecture project. We then add one pattern parameter, `TargetClass` that accepts a UML class as argument. In the pattern hot-spot methods, we implement the behavior of the pattern such that when a UML class is supplied to `TargetClass` as an argument, the following UML artifacts will be created in the model as a result of the pattern application.

- A UML interface with all the public methods of the target class, and a realization relationship from the target class to the created interface
- A Proxy class with the stereotype `<<VirtualProxy>>` that implements the common interface
- A directed association from the Proxy class to the target class.
- Two attributes in the proxy class, the "id" and the reference to the real class. Figure 26 shows the outcome of the pattern application to a UML class called Element. Figure 25 shows the code for the `expand()` of this pattern.

```
public boolean expand(PatternParameterValue value) {

Class domainClass = (Class) value.getValue();
Interface domainInterface ;

if( domainClass.getPackage().getMember("I"+domainClass.getName())== null){
domainInterface =
domainClass.getPackage().createOwnedInterface("I"+domainClass.getName());
}
else{

domainInterface =
(org.eclipse.uml2.uml.Interface)domainClass.getPackage().getMember("I"+doma
Class.getName());
}



EList domainClassOperationList = domainClass.getAllOperations();
Iterator domainClassOperationListIterator =
domainClassOperationList.iterator();

while(domainClassOperationListIterator.hasNext()){

Operation operation = (Operation) domainClassOperationListIterator.next();

if((operation.getVisibility()== VisibilityKind.PUBLIC_LITERAL)&&
(operation.getName().equalsIgnoreCase(domainClass.getName())== false)){

domainInterface.getOwnedOperations().add(operation);

}
}



if(domainClass.getInterfaceRealization(null, domainInterface)== null){

domainClass.createInterfaceRealization(null, domainInterface);
}

Model umlLibrary = (Model)
load(URI.createURI(org.eclipse.uml2.uml.resource.UMLResource.JAVA_PRIMITIVE
YPES_LIBRARY_URI));

PrimitiveType longPrimitiveType = (PrimitiveType)
umlLibrary.getOwnedType("long");



if (ProfileResource != null) {
List contents = ProfileResource.getContents();
if (contents.size() == 1 && contents.get(0) instanceof Profile) {
poeaaProfile = (Profile) contents.get(0);
}
}

if(domainClass.getPackage().getMember(domainClass.getName()+"Proxy")== null
{

Class proxyClass =
domainClass.getPackage().createOwnedClass(domainClass.getName()+"Proxy",
false);

domainClass.getModel().applyProfile(poeaaProfile);

proxyClass.applyStereotype(poeaaProfile.getOwnedStereotype("VirtualProxy",
true, false));


proxyClass.createInterfaceRealization(null,domainInterface);
```

```
Property innerAttribute =
proxyClass.createOwnedAttribute("inner"+domainClass.getName(), domainClass)
innerAttribute.setVisibility(VisibilityKind.PRIVATE_LITERAL);


Property idAttribute = proxyClass.createOwnedAttribute("id",
longPrimitiveType);
idAttribute.setVisibility(VisibilityKind.PRIVATE_LITERAL);

EList ownedParameterNames = new BasicEList();
EList ownedParameterTypes = new BasicEList();

ownedParameterNames.add("id");
ownedParameterTypes.add(longPrimitiveType);

Operation constructor = proxyClass.createOwnedOperation(proxyClass.getName(
ownedParameterNames, ownedParameterTypes);
constructor.setVisibility(VisibilityKind.PUBLIC_LITERAL);


proxyClass.getOwnedOperations().add(constructor);

Operation innerOperation = UMLFactory.eINSTANCE.createOperation();
innerOperation.setVisibility(VisibilityKind.PRIVATE_LITERAL);
innerOperation.setName("getInner"+domainClass.getName());
innerOperation.createReturnResult(domainClass.getName().toLowerCase(),domai
lass);


proxyClass.getOwnedOperations().add(innerOperation);

}



return true;
}
```

Figure 25: Code for `expand()` of *LazyLoadWithVirtualProxy* pattern

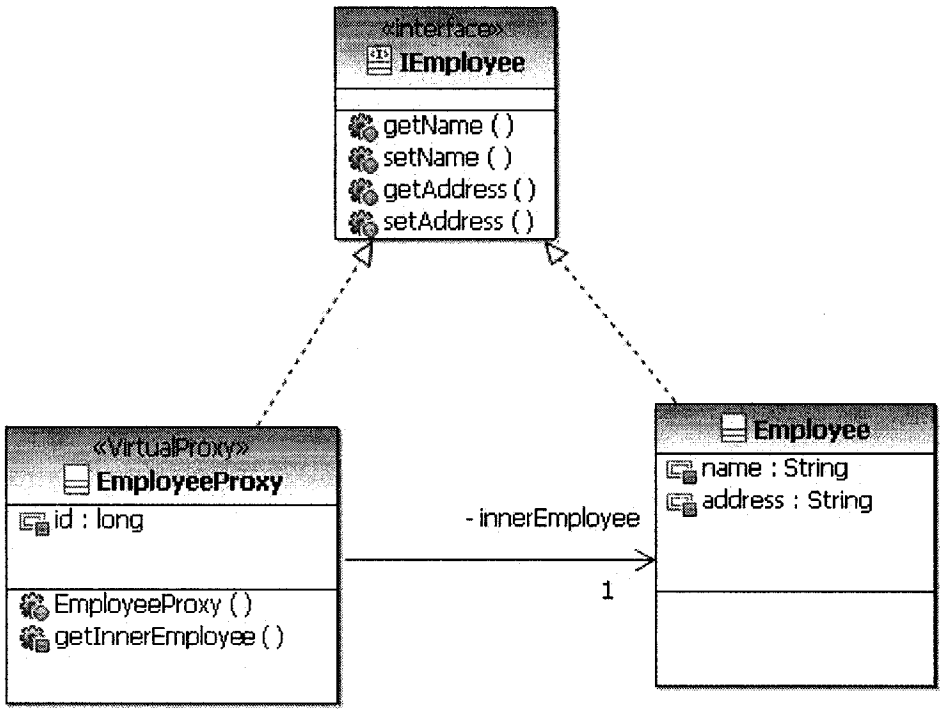

Figure 26: LazyLoadWithVirualProxy pattern applied to a UML class

4.4.3 Creating the Transformation

We created a separate JET `LazyLoadWithVirtualProxyTemplate` to generate code for the virtual proxy UML class. We added code to the *ClassRule* to invoke this template if it gets a UML class with the stereotype `<<VirtualProxy>>`. Complete JET source can be found in the Appendix.

4.5 Transformation 3: Finder

4.5.1 Simplifying Assumptions

The *Finder* class will contain two finder methods, one to find an object by "id" and the other to find all the objects in the database.

4.5.2 Implementing the Pattern

We create a *Finder* pattern inside the *PatternsOfEnterpriseApplicationArchitecture* project. We define one pattern parameter `Targetclass` that accepts a UML class. We implement the behavior of the pattern such that when a UML class is supplied to `TargetClass` as an argument, the following UML elements are added to the model:

- A UML *Finder* class for the target class
- Three attributes called `TABLE_NAME`, `SELECT_STRING` and `SELECT_ALL_STRING`, each of type String, are created in the *Finder* class. These attributes' values are set to the SQL statements that will be executed by the *Finder*.

Two methods, `findAll()` and `find(id)`, are also created inside the *Finder* Class. The `findAll()` method gets all the objects of the type of the target class from the database. Assuming that each object stored in the database has a unique id, the `find(id)` method

gets the object with the supplied id. Figure 28 shows the output of the application of the Finder pattern to the Employee UML class that is shown in Figure 26. Notice that as a result of the pattern application, the attributes of the Employee class are automatically plugged into the String values of SELECT_STRING and SELECT_ALL_STRING. An "id" and a "version" are also plugged in because each object that we store in the database has an id and a version. The Finder compares the id, the version and the values of other attributes in order to retrieve an object from the database. Figure 27 shows the code for the hot-spot method of the *Finder* pattern.

```
public boolean expand(PatternParameterValue value) {

Class domainClass = (org.eclipse.uml2.uml.Class) value.getValue();
Model model = domainClass.getModel();
Class finderClass = null;

Model umlLibrary = (Model) load(URI
.createURI(org.eclipse.uml2.uml.resource.UMLResource.UML_PRIMITIVE_TYPES_LIBI
RY_URI));

PrimitiveType stringPrimitiveType = (PrimitiveType)
umlLibrary.getOwnedType("String");


Model javaLibrary = (Model) load(URI
.createURI(org.eclipse.uml2.uml.resource.UMLResource.JAVA_PRIMITIVE_TYPES_LI
ARY_URI));

PrimitiveType longPrimitiveType = (PrimitiveType)
javaLibrary.getOwnedType("long");



Type resultSet = UMLFactory.eINSTANCE.createDataType();
resultSet.setName("ResultSet");

Package datasourcePackage;

if(model.getMember("datasource")== null){
datasourcePackage = model.createNestedPackage("datasource");

}else{

datasourcePackage = model.getNestedPackage("datasource");
}

if(datasourcePackage.getMember(domainClass.getName()+"Finder")== null){
finderClass =
datasourcePackage.createOwnedClass(domainClass.getName()+"Finder",false);

/*
* get the profile and apply it to the model
*/

if (ProfileResource != null) {
```

```
List contents = ProfileResource.getContents();
if (contents.size() == 1 && contents.get(0) instanceof Profile) {
poeaaProfile = (Profile) contents.get(0);
}
}


model.applyProfile(poeaaProfile);
finderClass.applyStereotype(poeaaProfile.getOwnedStereotype("Finder", true ,
false));


EList ownedParameterNames = new BasicEList();
EList ownedParmeterTypes = new BasicEList();


ownedParameterNames.add("id");
ownedParmeterTypes.add(longPrimitiveType);



Property tableName = finderClass.createOwnedAttribute("TABLE_NAME",
stringPrimitiveType);
tableName.setIsStatic(true);
tableName.setIsLeaf(true);
tableName.setVisibility(VisibilityKind.PRIVATE_LITERAL);
tableName.setStringDefaultValue("DbRegistry.getTablePrefix()+\""+domainClass.
etName()+"\"");


Property selectString = finderClass.createOwnedAttribute("SELECT_STRING",
stringPrimitiveType);
selectString.setIsStatic(true);
selectString.setIsLeaf(true);
selectString.setVisibility(VisibilityKind.PRIVATE_LITERAL);
selectString.setStringDefaultValue(StringUtility.getValueForSelect(domainClas
));



Property selectAllString =
finderClass.createOwnedAttribute("SELECT_ALL_STRING", stringPrimitiveType);
selectAllString.setIsStatic(true);
selectAllString.setIsLeaf(true);
selectAllString.setVisibility(VisibilityKind.PRIVATE_LITERAL);
selectAllString.setStringDefaultValue(StringUtility.getValueForFindAll(domair
lass));



Operation findAllOperation = UMLFactory.eINSTANCE.createOperation();
findAllOperation.setName("findAll");
findAllOperation.setVisibility(VisibilityKind.PUBLIC_LITERAL);
findAllOperation.setIsStatic(true);
findAllOperation.createReturnResult("rs", resultSet);
finderClass.getOwnedOperations().add(findAllOperation);




Parameter findParameter = UMLFactory.eINSTANCE.createParameter();
findParameter.setName("id");
findParameter.setType(longPrimitiveType);


Operation findOperation = UMLFactory.eINSTANCE.createOperation();
findOperation.setName("find");
findOperation.setVisibility(VisibilityKind.PUBLIC_LITERAL);
findOperation.setIsStatic(true);
findOperation.createReturnResult("rs", resultSet);
findOperation.getOwnedParameters().add(findParameter);
```

```
finderClass.getOwnedOperations().add(findOperation);
}
return true;
}
```

Figure 27: `expand()` of the Finder pattern

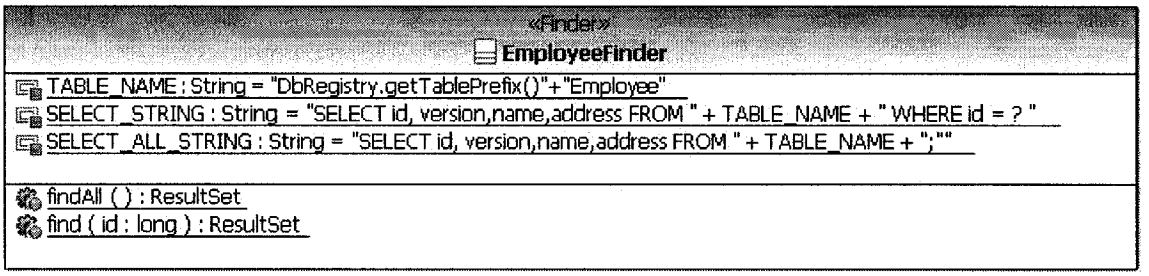

Figure 28: Output of the Finder pattern when applied to a UML class named Employee

4.5.3 Creating the Transformation

We create a JET `FinderTemplate` to generate code for the *Finder* class. We add code to the ClassRule to invoke this template if it gets a UML class with a stereotype `<<Finder>>`. Complete JET source can be found in the Appendix.

4.6 Summary

In this chapter, we demonstrated the process of authoring a subset of the PoEAA patterns. We created model transformations to generate pattern code. We explained the rationale behind our design decisions, the working of our transformations and the output of pattern application with the help of small examples. In the next chapter, we make use of our model transformations in the development of a Team Registration System.

5 The Case Study: Team Registration System

In the previous chapter, we authored a subset of PoEAA patterns and created model transformations to generate pattern code. In this chapter, we validate our work by applying our authored patterns and the model transformations in the development of a Team Registration System.

5.1 Team Registration System: Introduction and Domain Model

The Team Registration System (TRS) is a small enterprise application. TRS is used to form and manage student teams for the purpose of course work. TRS can be best explained with the help of a domain model that is created after an initial domain analysis. The domain model for TRS is shown in Figure 29. The domain concepts that are presented include the following:

- **Student**: A university student.
- **Student Registry**: The central registry that contains the information of all students in the institution. There is only instance of the Student Registry.
- **Course Offering**: A university course which is being offered over a given year and term(s).
- **Team**: A group of 4-6 students. Teams are of two types, confirmed and non-confirmed.
- **Approver:** uses the system to approve a non-confirmed team.
- **Instructor:** Instructor for a course offering.

Two fundamental concepts in the Team Registration System are **Student** and **Course Offering**. As can be seen in Figure 29, a **Student** can be registered in many **Course Offerings** and a **Course Offering** will hold a **class list** consisting of many **Students**.

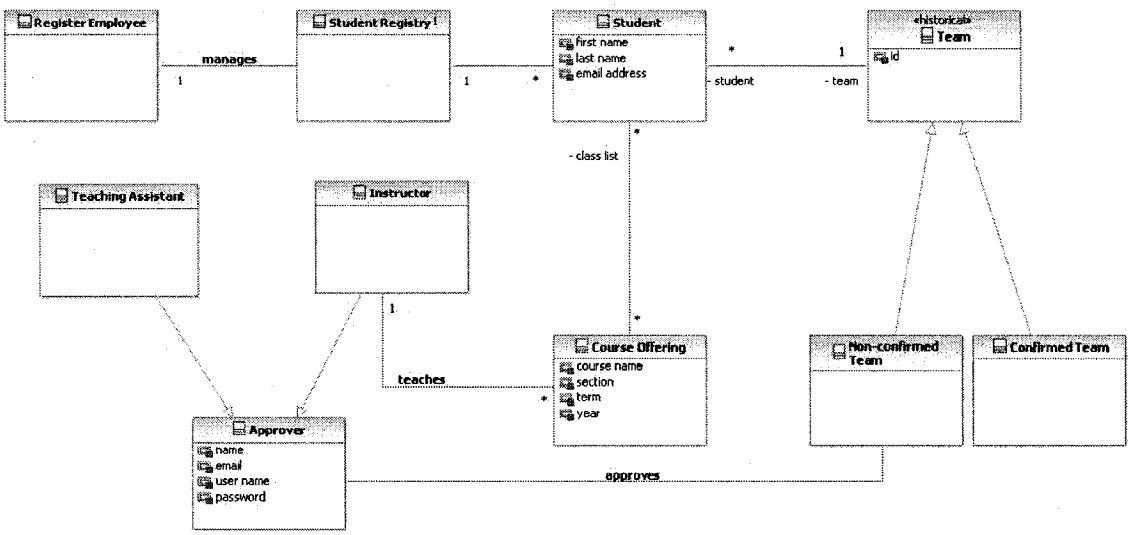

Figure 29: Team Registration System Domain Model

For the purpose of this case study, we choose to implement a simplified subset of the required features. The feature subset chosen is adding and removing students from a course offering.

5.2 RSA Project Setup

In order to start the development, we create two separate projects in the RSA workspace, a UML modeling project *TeamRegistrationSystemModel* and a Java project *TeamRegistrationSystemCode* that contains all the code for the project. Figure 30 shows the two projects set up in the RSA Project Explorer view.

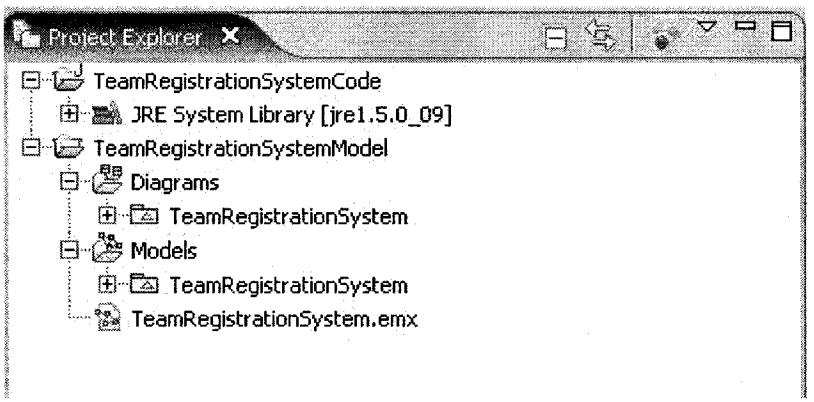

Figure 30: Team Registration System Project Setup

5.3 First Features: Add/Remove Students from CourseOffering

We start the development for the following two chosen features

- Add a **Student** to the class list
- Remove a **Student** from the class list

As MDD practitioners, we first create a UML design model that is inspired from TRS domain model. We create a separate UML package for domain model layer called `domain` inside the TRS model. Two classes, *Student* and *CourseOffering*, that are required for the current feature set, are shown in Figure 31.

A design decision to make is which class should know about the addition or removal of student(s) registered in a course offering? According to the "Information Expert" principle [Larman 2005], responsibility should be assigned to a class that has the information required to fulfill the responsibility. By Information Expert principle, *CourseOffering* should add and remove students. Hence we add the two methods `addStudent()` and `removeStudent()` to class *CourseOffering*.

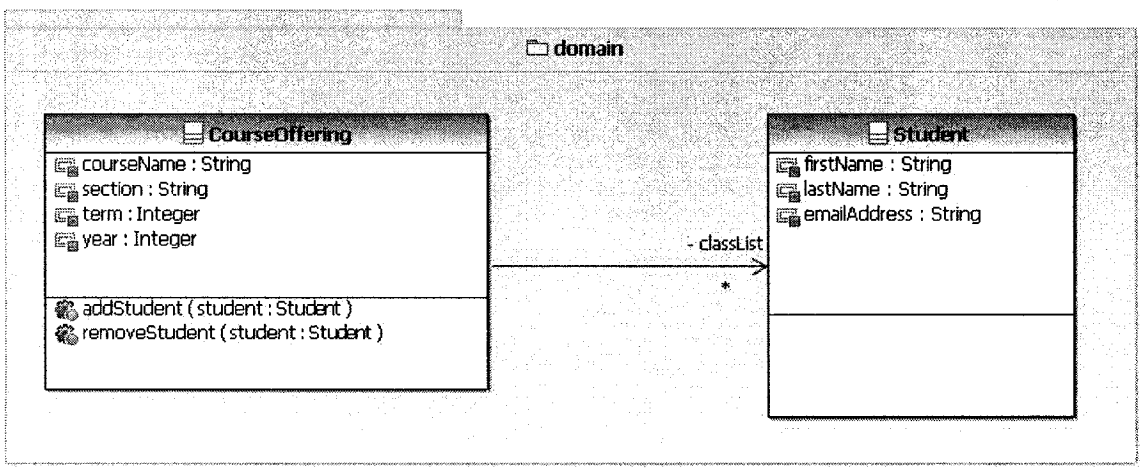

Figure 31: TRS class diagram for addition or removal of Students from CourseOffering

In addition to `addStudent()` and `removeStudent()`, we need setter and getter methods for attributes of the two classes. RSA does not provide a mechanism to create getter and setter methods in model classes. One work-around is to use the UML-to-Java transformation with the "Generate getters and setters" option turned on to generate getters and setters in the code and then reverse engineer to introduce the getters and setters into

the model. However, we create a *GetterSetter* pattern in RSA to create the getters and setters in the model itself. The *GetterSetter* pattern has three pattern parameters, `TargetClass`, `AttributesWithGetters` and `AttributesWithSetters`. `TargetClass` accepts a UML class for which the getters and setters are to be created. `AtributesWithGetters` and `AttributesWithSetters` both accept zero or more UML attributes that belong to the `TargetClass`. Figure 32 shows the two pattern instances of the *GetterSetter* pattern, the supplied arguments, and the generated getters and setters in the UML classes. Notice that we do not create a setter method for `classList`.A setter for the property `classList` would allow a client of the *CourseOffering* class to manipulate the contents of `classList` without the knowledge of the *CourseOffering* class. Instead, `classList` should only be manipulated using the `addStudent()` and `removeStudent()` operations. Additionally, a client of *CourseOffering* should only be allowed to iterate over the collection without being able to modify the collection. Hence, we create a method `CourseOffering.getClassList()` that returns a non-modifiable view of the `classList`.

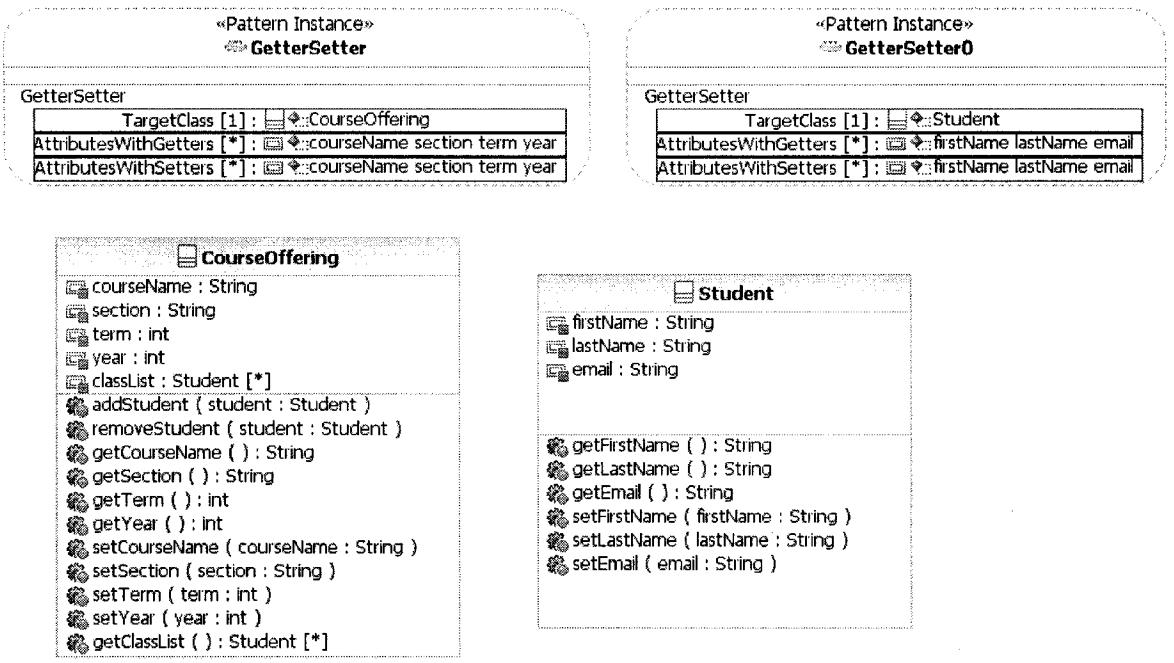

Figure 32:*GettterSetter* **pattern applied to** *CourseOffering* **and** *Student*

After we have created all the required methods in the model, we make use of the UML-to-Java transformation to generate code for the two classes. In the generated code, we first make the two classes extend the *DomainObject* class that is part of the SoenEA framework. *DomainObject* class contains the generic code for assigning ids and version numbers to domain classes. The SoenEA framework uses a version number for each domain class to handle concurrency management.

We also add the code for the methods with empty method bodies. For example, the code that is enclosed in a in Figure 33 shows the code that is added to the `getClassList():Student[*]` method of *CourseOffering* class.

```
  .....
import org.dsrg.dom.DomainObject;

public class CourseOffering extends DomainObject {
.......

   /**
    * <!-- begin-user-doc -->
```

```
    * <!-- end-user-doc -->
    * @return
    * @generated "UML to Java V5.0
(com.ibm.xtools.transform.uml2.java5.internal.UML2JavaTransform)"
    */


   public Collection<Student> getClassList() {
      // begin-user-code

      return Collections.unmodifiableSet(classList);
      // end-user-code
   }
}
```

Figure 33: Behavior added to empty method bodies for `addStudent()` and `removeStudent()` generated by way of UML-to-Java transformation

In order to introduce the changes made in the code back into the model, we reverse engineer using the "Java-to-UML" transformation. Figure 34 shows the domain package in the TRS model after we reverse engineer the code.

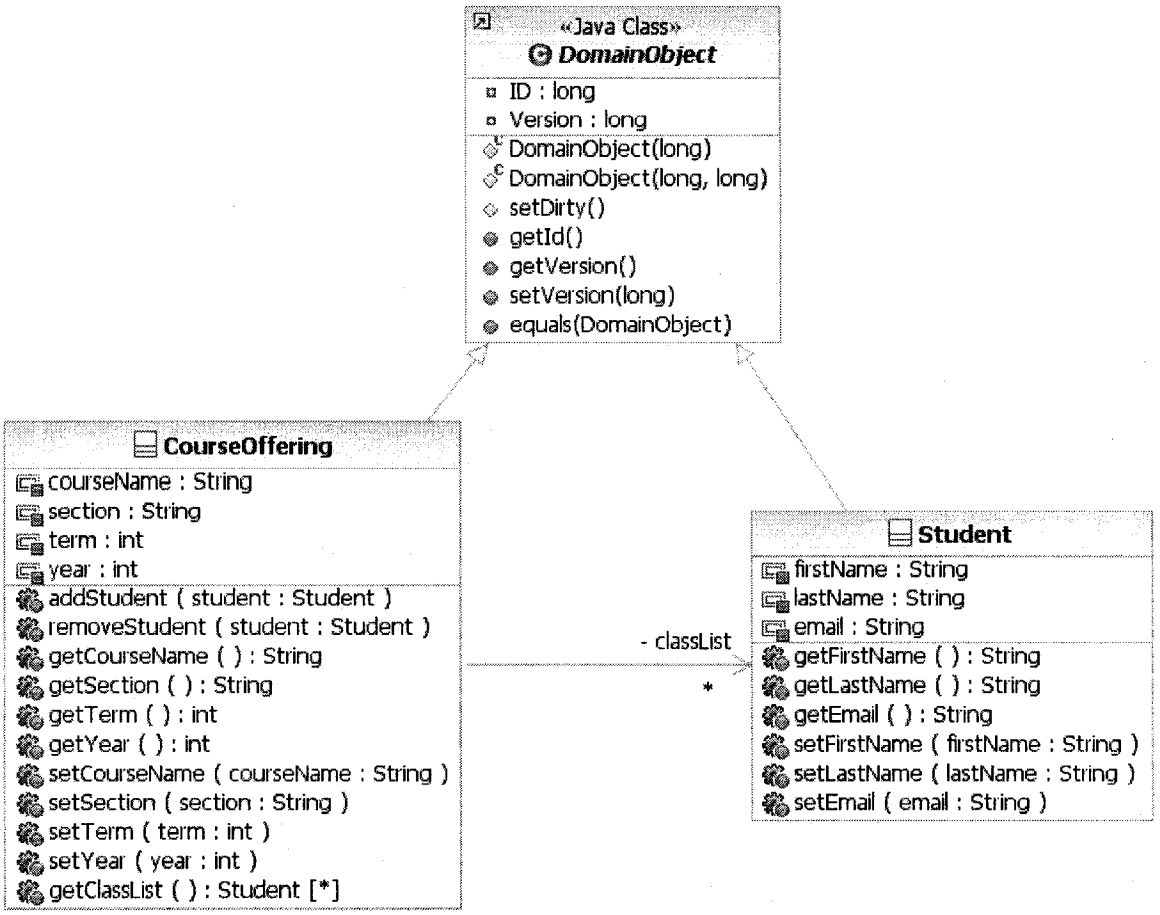

Figure 34: Class Diagram showing the model after reverse engineering

5.4 Applying *LazyLoadWithVirtualProxy* Pattern

When a *CourseOffering* object is loaded into memory, we do not want to load all the *Student* objects that are present in the `CourseOffering.classList` collection. Thus, we make use of our *LazyLoadWithVirtualProxy* pattern to create a *StudentProxy* class and the *IStudent* interface (See - Figure 35). In addition, the *CourseOffering* class is modified to refer to the *IStudent* interface since the *CoursOffering* class can now contain instances of *StudentProxy* too. The required changes are shown in Figure 35 in the CourseOffering class.

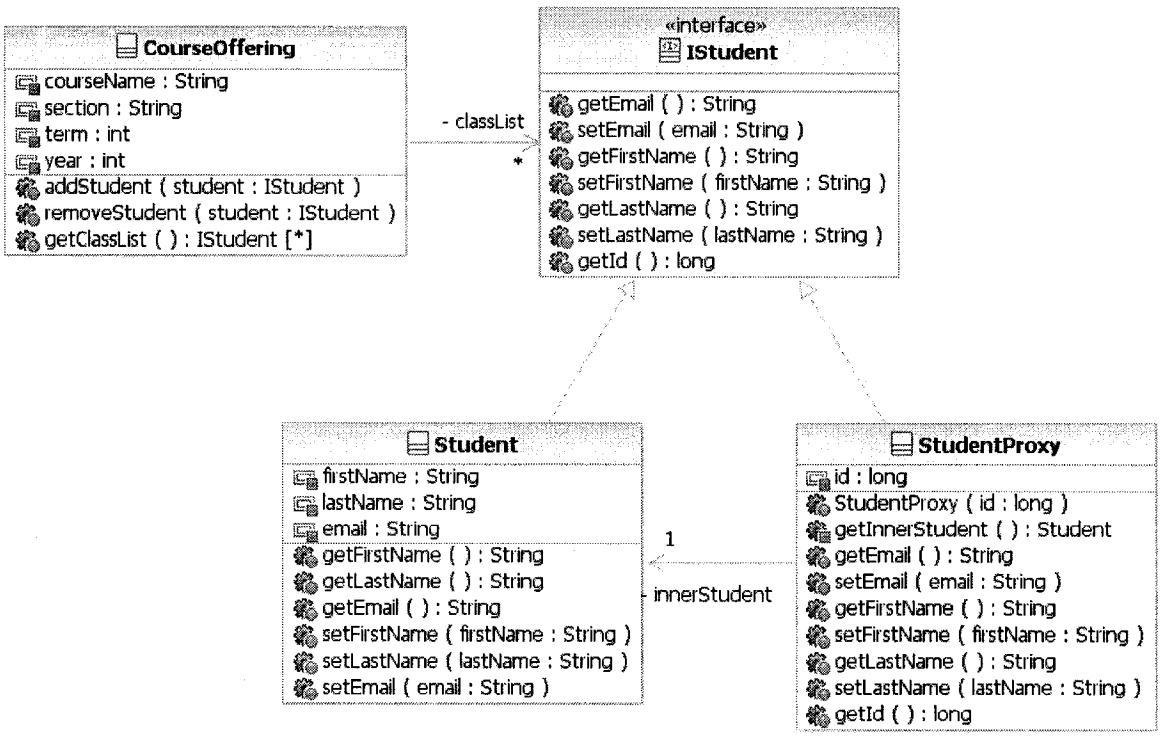

Figure 35: Class Diagram showing output of *LazyLoadWithVirtualProxy* Pattern

After adding the *IStudent* and *StudentProxy* to the model, we generate the corresponding code using the 'PoEAA-to-Java' transformation. The generated code for the *StudentProxy* is shown in Figure 36.

```
package domain;

import org.dsrg.dom.ProxyException;


public class StudentProxy implements IStudent {

    private Student innerStudent;
    private long id;

    public StudentProxy(long id) {
        this.id = id;
    }

    private synchronized Student getInnerStudent() {
        if(innerStudent==null)
            try {
                innerStudent = StudentMapper.map (id);
            } catch (Exception e) {
                throw new ProxyException(e);
            }
        return innerStudent;

    }
```

```
    public String getEmail() {
        return getInnerStudent().getEmail();
    }

    public void setEmail(String email) {
        getInnerStudent().setEmail(email);

    }

    public String getFirstName() {
        return getInnerStudent().getFirstName();

    }

    public void setFirstName(String firstName) {
        getInnerStudent().setFirstName(firstName);
    }
    public String getLastName() {
        return getInnerStudent().getLastName();
    }

    public void setLastName(String lastName) {
        getInnerStudent().setLastName(lastName);
    }

    public long getId() {
        return id;
    }


}
```

Figure 36: ***StudentProxy*** **code generated by way of PoEAA-to-Java transformation**

5.5 Applying *TableDataGatewayWithDataMapper* and *Finder* Pattern

Enterprise applications usually involve data that needs to be persisted. In the TRS, the domain objects *Student* and *CourseOffering* are to be persisted in the database. We create the corresponding tables TRS_*CourseOffering* and *TRS_Student* in the database. There is a many-to-many association between course offerings and students. Therefore, a third table *TRS_CourseOffering_TRS_Student* is created to contain this many-to-many association.

After creating the database tables, we create instances of the *TableDataGatewayWithDataMapper* pattern, one for each domain class. The arguments for the pattern instance are the domain class and the class attributes that are to be persisted (See - Figure 37). Notice that we do not supply the `classList` attribute of the

CourseOffering class to the pattern parameter `Attributes`. As mentioned in Section 4.3.1, this is a limitation of our automated pattern. The association between a Course Offering and its students should be explicitly implemented manually in code after forward transformation

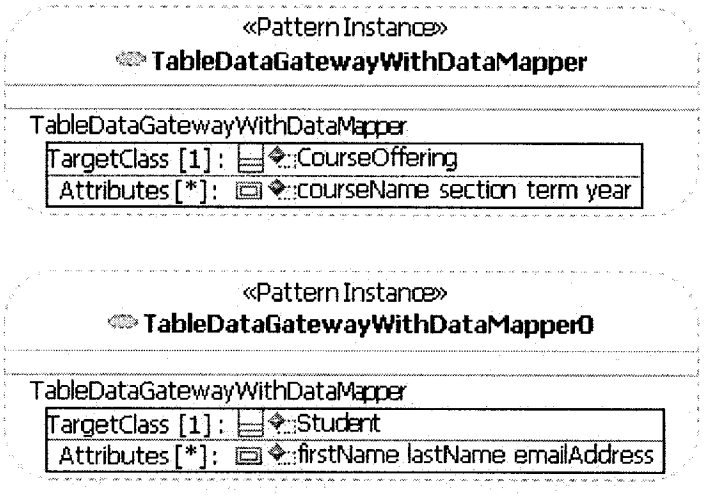

Figure 37: Pattern instances for Student and CourseOffering classes

The application of the pattern to the two classes results in the creation of a TDG UML class and a DM UML class for both *Student* and *CourseOffering* (See - Figure 38). Notice that the required attributes and operations in the TDG and the DM UML classes are also auto-generated.

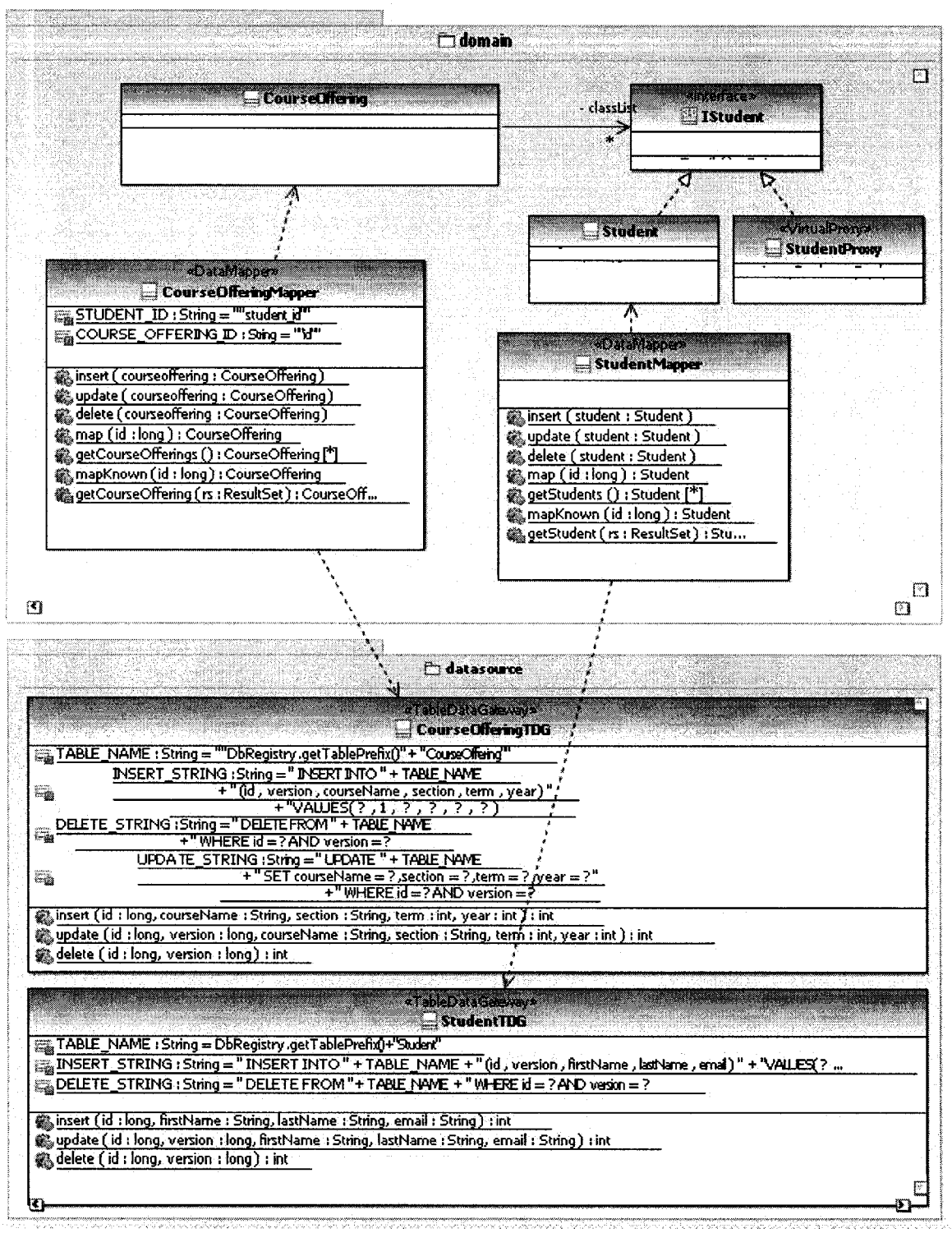

Figure 38: Class Diagram showing Data Mappers and TDGs

After generating the Data Mappers and the TDGs, we apply our Finder pattern to the *Student* and the *CourseOffering* class to generate their *Finder* classes. Figure 39 shows the two *Finder* pattern instances and the supplied arguments.

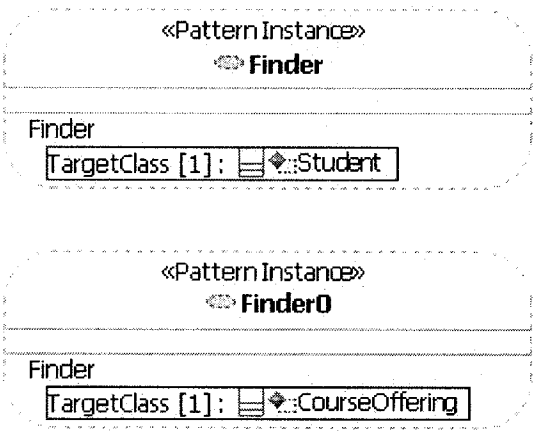

Figure 39: Pattern instances for *Finder* pattern with supplied arguments

Figure 40 shows the two Finder classes in the `datasource` package in the TRS model after the application of the Finder pattern.

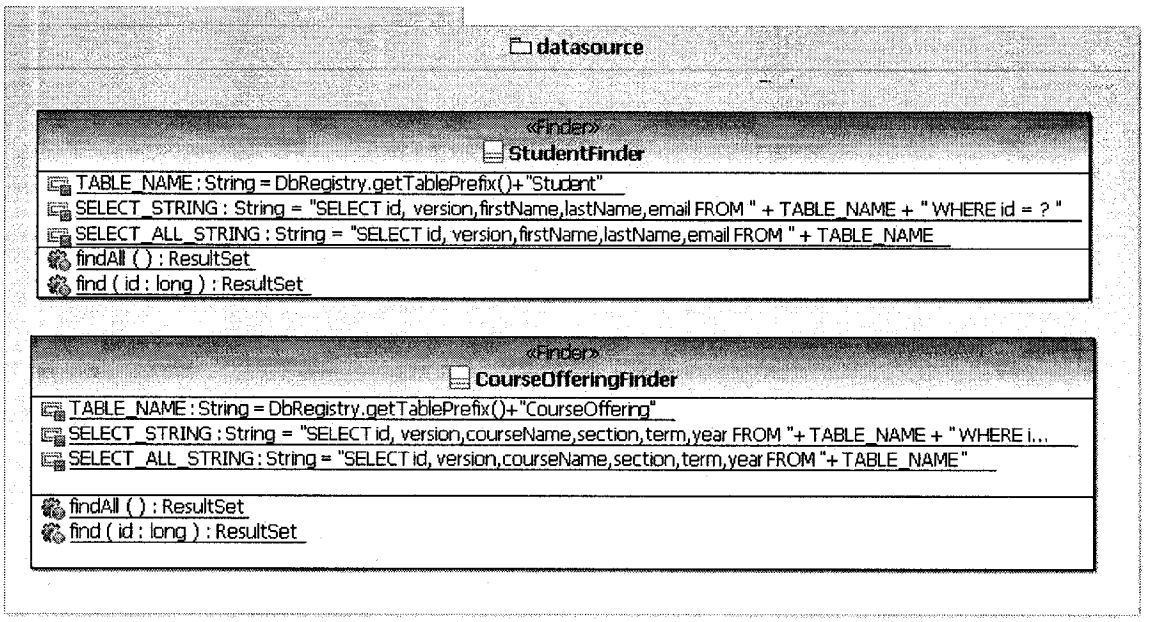

Figure 40: Class Diagram showing the *Finder* Classes

After we apply the *TableGatewayWithDataMapper* and the *Finder* patterns to the model classes, our next step is to use the PoEAA-to-Java transformation to generate pattern code.

This generated code is not fully functional. Our transformation does not take care of mapping the associations between *CourseOffering* and its `classList`. The code for managing the mappings has to be filled explicitly. First, when a course offering is loaded, along with the class attributes like `courseName`, the course offering's student class list should also be loaded. We have made the design decision of creating student proxies instead of loading the entire student information. For this purpose, we only load the ids of the students in a class list for a particular course offering using `CourseMapperFinder.-findStudentIds(courseOfferingId):Long[*]`. These ids are used to create *StudentProxy* instances. Once all the values for the course offering are loaded, we create an instance of *CourseOffering* with the constructor `CourseOffering(id, version,-`

courseName,section,term,year,students). Note that the students parameter is of type IStudent[*].

In the *CourseOfferingMapper*, when inserting an instance of *CourseOffering*, the associations between the *CourseOffering* instance and its *Student* instances are also inserted. This is done by the method CourseOfferingMapper.insertStudent-Associations(courseoffering).This method will extract all the student ids in the classList and invoke the method CourseOfferingTDG.insertStudent-Associations(courseOfferingId,studentIds). The *CourseOfferingTDG* then inserts the required mapping in the *TRS_CourseOffering_TRS_Student* table.

For updating a course offering, we cannot determine the students deleted from the course offerings class list. Therefore, we first delete all the associations in the *TRS_CourseOffering_TRS_Student* table for a given course offering. We then insert the latest class list associations. The previous class list associations are deleted using CourseOfferingMapper.deletePreviousStudentAssociations(courseoffering) and CourseOfferingTDG.deleteCourseOfferingStudentAssociations(course-OfferingId). For re-inserting the associations, we reuse the functionality Course-OfferingTDG.insertStudentAssociations(courseOfferingId,studentIds)

The generated code for the *CourseOfferingTDG*, *CourseOfferingMapper* and *CourseFinder* are shown in Figure 41, Figure 42 and Figure 43 respectively. The code enclosed in boxes indicates the additional code inserted manually to manage the associations.

```
package datasource;

import java.util.Set;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
```

```
import java.sql.SQLException;
import java.sql.Connection;

import org.dsrg.ts.UniqueIdFactory;
import org.dsrg.ts.threadLocal.DbRegistry;

public class CourseOfferingTDG {

   private static final String TABLE_NAME = DbRegistry.getTablePrefix()+
"CourseOffering";

   private static final String INSERT_STRING = " INSERT INTO " + TABLE_NAM
         + " (id , version , courseName , section , term , year ) "
         + "VALUES( ?  , 1 ,  ?  ,  ?  ,  ?  ,  ?  );";

   private static final String DELETE_STRING = " DELETE FROM " + TABLE_NAM
         + " WHERE id = ? AND version = ? ;";

   private static final String UPDATE_STRING = " UPDATE  " + TABLE_NAME
         + " SET courseName = ? ,section = ? ,term = ? ,year = ? "
         + " WHERE id = ? AND version = ?;";

   private static final String INSERT_STUDENT_ASSOCIATION = "INSERT INTO
TRS_CourseOffering_TRS_Student(courseOffering_id,student_id)
VALUES(?,?);";

   private static final String DELETE_STUDENT_ASSOCIATION = "DELETE FROM
TRS_CourseOffering_TRS_Student WHERE courseOffering_id=?;";

   public static int insert(long id, String courseName, String section,
         int term, int year) throws SQLException {

      Connection con = DbRegistry.getDbConnection();
      PreparedStatement ps = con.prepareStatement(INSERT_STRING);
      ps.setLong(1, id);
      ps.setString(2, courseName);
      ps.setString(3, section);
      ps.setInt(4, term);
      ps.setInt(5, year);
      return ps.executeUpdate();
   }

   public static int update(long id, long version, String courseName,
         String section, int term, int year) throws SQLException {

      Connection con = DbRegistry.getDbConnection();
      PreparedStatement ps = con.prepareStatement(UPDATE_STRING);
      ps.setLong(5, id);
      ps.setLong(6, version);
      ps.setString(1, courseName);
      ps.setString(2, section);
      ps.setInt(3, term);
      ps.setInt(4, year);
      return ps.executeUpdate();
   }

   public static int delete(long id, long version) throws SQLException {

      deleteCourseOfferingStudentAssociations(id);

      Connection con = DbRegistry.getDbConnection();
      PreparedStatement ps = con.prepareStatement(DELETE_STRING);
      ps.setLong(1, id);
      ps.setLong(2, version);
      return ps.executeUpdate();


   }


   /**
```

```
     * <!-- begin-user-doc -->
     * <!-- end-user-doc -->
     * @param courseOfferingId
     * @param studentIds
     * @throws SQLException
     * @generated "UML to Java V5.0
(com.ibm.xtools.transform.uml2.java5.internal.UML2JavaTransform)"
     */

    public static void insertStudentAssociations(Long courseOfferingId,
            Long... studentIds) throws SQLException {
        // begin-user-code

        Connection con = DbRegistry.getDbConnection();
        PreparedStatement ps =
con.prepareStatement(INSERT_STUDENT_ASSOCIATION);

        for(Long studId:studentIds){
            ps.setLong(1, courseOfferingId);
            ps.setLong(2, studId);

            ps.addBatch();
        }

        ps.executeBatch();
        // end-user-code
    }


    public static void deleteCourseOfferingStudentAssociations(long
courseOfferingId) throws SQLException {

        Connection con = DbRegistry.getDbConnection();
        PreparedStatement ps =
con.prepareStatement(DELETE_STUDENT_ASSOCIATION);

        ps.setLong(1, courseOfferingId);
        ps.executeUpdate();

    }

}
```

Figure 41: CourseOfferingTDG generated by way of *TableDataGatewayWithDataMapper* pattern with added code

```
package domain;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Collection;
import java.util.LinkedHashSet;
import java.util.List;

import org.dsrg.UoW.ObjectRemovedException;
import org.dsrg.UoW.UoW;
import org.dsrg.dom.DomainObjectCreationException;
import org.dsrg.dom.GenericMapper;
import org.dsrg.dom.MapperException;
import org.dsrg.dom.MetaDomainObject;

import datasource.CourseOfferingFinder;
import datasource.CourseOfferingTDG;

public class CourseOfferingMapper implements GenericMapper<CourseOffering
{

   private static final String STUDENT_ID = "student_id";

   private static final String COURSE_OFFERING_ID = "id";

   public void insert(CourseOffering courseoffering) throws MapperExceptic
{
      try {
         CourseOfferingTDG.insert(courseoffering.getId(), courseoffering
                        .getCourseName(), courseoffering.getSection(),
                        courseoffering.getTerm(), courseoffering.getYear()


         insertStudentAssociations(courseoffering);

      } catch (SQLException e) {
         throw new MapperException(e);
      }
   }

   public void update(CourseOffering courseoffering) throws MapperExceptic
{
      try {

         CourseOfferingTDG.update(courseoffering.getId(), courseoffering
                        .getVersion(), courseoffering.getCourseName(),
                        courseoffering.getSection(),
courseoffering.getTerm(),
                        courseoffering.getYear());


         //delete previous associations of coid,
         //we are deleting all associations because we dont know which
associations have been removed and which are added
         // so we remove everything and add everything afresh

         deletePreviousStudentAssociations(courseoffering);

         //                      insert new assocations of coid
         insertStudentAssociations(courseoffering);
      } catch (SQLException e) {
         throw new MapperException(e);
      }
   }

   private void deletePreviousStudentAssociations(CourseOffering
courseoffering)
         throws MapperException {

      try {
```

```
            CourseOfferingTDG
    .deleteCourseOfferingStudentAssociations(courseoffering.getId());
        } catch (SQLException e) {
            e.printStackTrace();
            throw new MapperException(e);
        }

    }

    public void delete(CourseOffering courseoffering) throws MapperExceptic
{
        try {
            CourseOfferingTDG.delete(courseoffering.getId(), courseoffering
                        .getVersion());
        } catch (SQLException e) {
            throw new MapperException(e);
        }
    }

    public static CourseOffering map(long id) throws SQLException,
            DomainObjectCreationException {
        try {
            return mapKnown(id);
        } catch (DomainObjectCreationException e) {
            System.out.println(e.getMessage());
        } catch (ObjectRemovedException e) {
            System.out.println(e.getMessage());
        }
        ResultSet rs = CourseOfferingFinder.find(id);
        if (!rs.next())
            throw new DomainObjectCreationException("Does not exist Mapper");
        return getCourseOffering(rs);
    }

    public static List<CourseOffering> getCourseOfferings() throws
SQLException {
        ResultSet rs = CourseOfferingFinder.findAll();
        List<CourseOffering> list = new ArrayList<CourseOffering>();
        while (rs.next()) {
            list.add(getCourseOffering(rs));
        }
        return list;
    }

    public static CourseOffering mapKnown(long id)
            throws DomainObjectCreationException, ObjectRemovedException {
        MetaDomainObject<CourseOffering> mdo = new
MetaDomainObject<CourseOffering>(
                    CourseOffering.class, id);
        if (UoW.getCurrent().hasObject(mdo)) {
            return (CourseOffering) UoW.getCurrent().getObject(mdo);
        }
        throw new DomainObjectCreationException("Does not exist");
    }

    private static void insertStudentAssociations(CourseOffering
courseOffering)
            throws SQLException {
        // begin-user-code

        Collection<IStudent> students = courseOffering.getClassList();

        // create a collection of student ids
        Collection<Long> studIds = new LinkedHashSet<Long>();
        for (IStudent stud : students) {
            studIds.add(stud.getId());
        }

        // TDG.insert Mapping
        Long[] studentIdArray = (Long[]) studIds.toArray(new
Long[studIds.size()]);
        CourseOfferingTDG.insertStudentAssociations(courseOffering.getId(),
```

```
                studentIdArray);

        // end-user-code
    }

    private static CourseOffering getCourseOffering(ResultSet rs)
            throws SQLException {

        // find student ids for this course offering
        Long courseOfferingId = rs.getLong(COURSE_OFFERING_ID);
        ResultSet studentIdSet = CourseOfferingFinder
                .findStudentIds(courseOfferingId);

        // create student proxies
        Collection<IStudent> studentProxies =
createStudentProxies(studentIdSet);

        // create a new CO giving it proxies.
        CourseOffering courseoffering = new CourseOffering(rs
                .getLong(COURSE_OFFERING_ID), rs.getLong("version"), rs
                .getString("courseName"), rs.getString("section"), rs
                .getInt("term"), rs.getInt("year"), studentProxies);

        UoW.getCurrent().registerClean(courseoffering);
        return courseoffering;
    }

    private static Collection<IStudent> createStudentProxies(
            ResultSet studentIdSet) {

        //          for each student in resultset
        Collection<IStudent> students = new LinkedHashSet<IStudent>();
        try {

            while (studentIdSet.next()) {
                Long id = studentIdSet.getLong(STUDENT_ID);
                StudentProxy proxy = new StudentProxy(id);
                students.add(proxy);

            }

        } catch (SQLException e) {

            e.printStackTrace();
        }

        return students;

    }

}
```

Figure 42: *CourseOfferingMapper* generated by way of *TableDataGatewayWithDataMapper pattern* with added code.

```
package datasource;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import org.dsrg.ts.threadLocal.DbRegistry;

public class CourseOfferingFinder {

   private final static String TABLE_NAME = DbRegistry.getTablePrefix()
         + "CourseOffering";

   private final static String SELECT_STRING = "SELECT id,
version,courseName,section,term,year FROM "
         + TABLE_NAME + " WHERE id = ? ";

   private final static String SELECT_ALL_STRING = "SELECT id,
version,courseName,section,term,year FROM "
         + TABLE_NAME + ";";

   private static final String SELECT_STUDENT_IDS = "SELECT student_id FRO
TRS_CourseOffering_TRS_Student WHERE courseOffering_id=?;";

   public static ResultSet findAll() throws SQLException {
      Connection con = DbRegistry.getDbConnection();
      PreparedStatement ps = con.prepareStatement(SELECT_ALL_STRING);
      return ps.executeQuery();
   }

   public static ResultSet find(long id) throws SQLException {
      Connection con = DbRegistry.getDbConnection();
      PreparedStatement ps = con.prepareStatement(SELECT_STRING);
      ps.setLong(1, id);
      return ps.executeQuery();
   }

   /**
    * <!-- begin-user-doc -->
    * <!-- end-user-doc -->
    * @param courseOfferingId
    * @return
    * @throws SQLException
    * @generated "UML to Java V5.0
(com.ibm.xtools.transform.uml2.java5.internal.UML2JavaTransform)"
    */
   public static ResultSet findStudentIds(long courseOfferingId) throws
SQLException {
      // begin-user-code


         Connection con = DbRegistry.getDbConnection();
         PreparedStatement ps = con.prepareStatement(SELECT_STUDENT_IDS);
         ps.setLong(1, courseOfferingId);
         return ps.executeQuery();

      // end-user-code
   }

}
```

Figure 43: *CourseOfferingFinder* generated by way of *Finder* pattern with added code.

The code that is added is reverse engineered back into the model. The update UML model is shown in Figure 44.

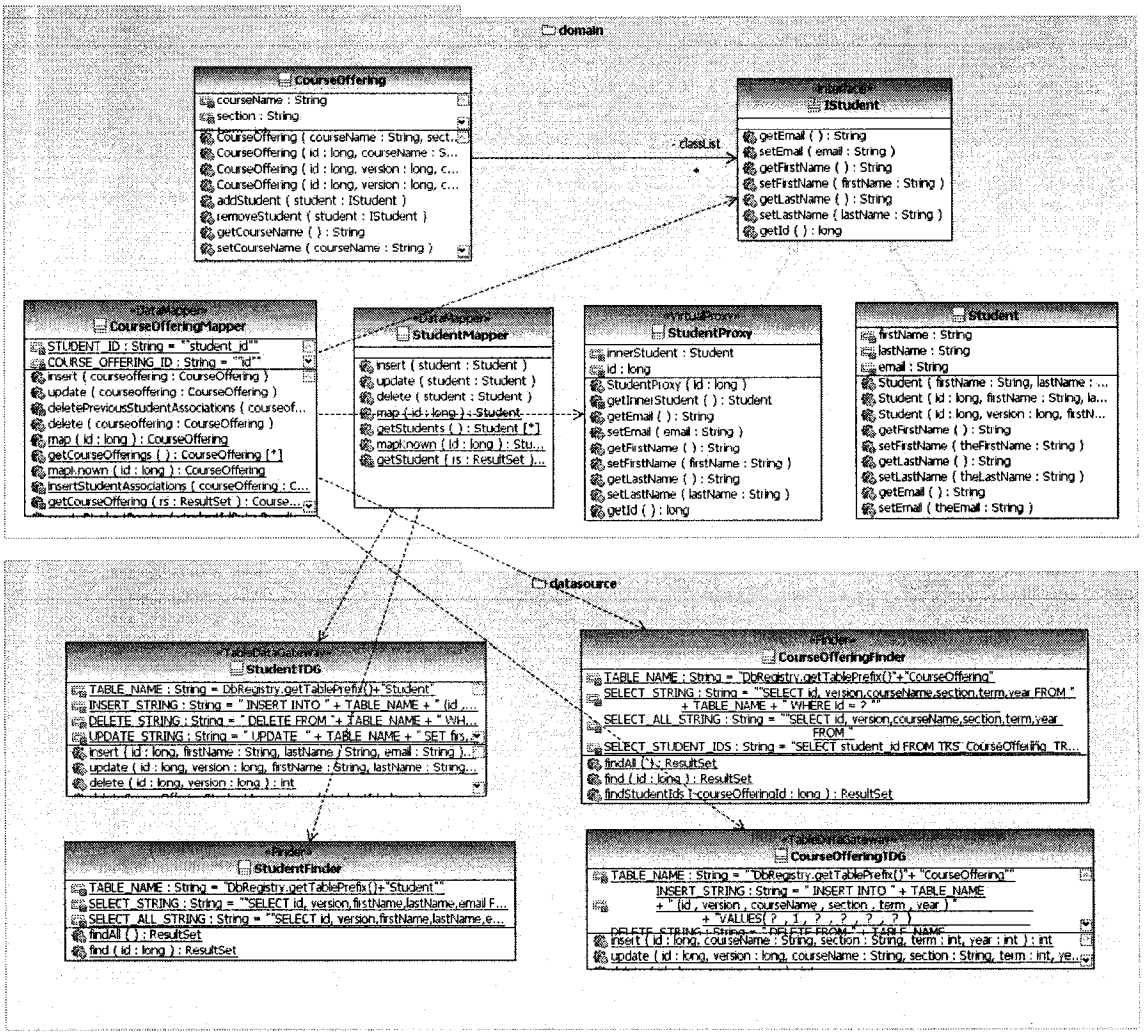

Figure 44: Class Diagram showing the model after reverse engineering

5.6 Testing the code

The code is verified with the help of unit tests. Using the JUnit framework, we check the following functionality of the generated code.

- View Course Offering
- Add Student to Course Offering
- Remove Student from Course Offering
- View Student

5.7 Summary

In this case study, we started with a domain model for the *Team Registration System*. We selected a small feature set and created a design model containing classes that were relevant to the feature set. We made use of our automated patterns and transformations to create pattern classes in the model and generate pattern code. We then added the remaining implementation details that were not auto-generated and reverse engineered the code back into the model. In the end, we validated our work using JUnit test cases.

6 Limitations

The main limitation of our work is that our code generation setup has a heavy dependency on the SoenEA framework. The developer who wants to make use of our automated patterns and transformations has to have a good understanding of the SoenEA framework. Because of this strong dependency, the auto-generated code is good for use with the SoenEA framework only.

The second main limitation of our transformations is that unlike Hibernate [Hibernate 2007], our transformations impose a restriction on the type and multiplicity of the UML attributes that a class can own. For example, 100% of the *Table Data Gateway (TDG)* code is generated for a UML class only if it has attributes of primitive types and the multiplicity of the attributes is equal to one. The transformations do not support collections. When a class has a UML association with another class, partial code is generated and the code to maintain the association between two classes in the database has to be written manually. Another major limitation of the transformation system is that code to persist collections is not auto-generated.

With changes in requirements, the *Domain Model* is bound to change. Every time we make a change in a UML class, we need to reapply the pattern to the class and regenerate the code for *Table Data Gateway*, *Data Mapper* and *Virtual Proxy*. The code that was previously generated is completely overwritten.

Like our transformations, Hibernate too deals with object-relational mapping. The limitations that exist in our transformations do not exist in Hibernate. Unlike our transformations, Hibernate does not have a dependency on any development framework.

Hibernate is a non-intrusive solution [Bauer & King 2007]. A developer does not have to follow any hibernate-specific rules while writing the business logic of the application. As opposed to our transformations, Hibernate is not restricted to attributes of primitive types only. It allows all data types. In addition, there is no restriction on the multiplicity of an attributes as Hibernate supports persistence of collections. With Hibernate, classes can use object composition and inheritance relationships. Hibernate also provides the auto-generation of proxies for persistent classes. It allows the developer to specify if a proxy should be created for a class. In short, Hibernate is a comprehensive object-relational mapping solution. The developer of a system can make full use of object orientation and focus on business logic without worrying about persistence.

It is quite evident that when compared to our transformations, Hibernate is a much superior alternative. However, in this thesis our focus was not to create a hibernate-like solution. The primary focus of this thesis was on code generation in the context of model-driven development. We chose a few enterprise patterns for the purpose of code generation and because these patterns deal with O/R mapping, a comparison with Hibernate is deemed necessary.

However, if we are to reconsider the design of our transformations, can we overcome some or all of the limitations? We believe that for a *Table Data Gateway*, we could generate 100% code if we used the database schema as input to the code-generation process. We could generate one complete *Table Data Gateway* for each table in the database and in that we will not to have to worry about complex relationships between objects and all of the code can be auto-generated.

However, generating complete *Data Mappers* is not straightforward since we cannot use database tables as input. The working of a *Data Mapper* is much more complex because one class may have to be mapped to more than one database tables and vice versa. For example, if a class contains a collection, there will be a separate table in the database for storing the collection and the *Data Mapper* will have to maintain the mapping. However if there is a one-to-one relationship between a class and a table, then all of the *Data Mapper* code can be auto-generated. If the associations are complex, basic skeleton for the *Data Mapper* may be auto-generated but the mappings will have to be inserted manually.

7 Conclusion and Future Work

Model transformations are the heart and soul of Model-Driven Development (MDD). The ability to auto-generate code from models is critical to the success and adoption of MDD in the Software Engineering industry. Through this thesis, we made an effort to demonstrate that modeling and code generation technology has matured to the point that significant amount of code can be auto-generated.

In this thesis, we have demonstrated that productivity can be improved significantly with the use of code generation. By automating software patterns, the problem of skeletal code generation can be overcome. Significant amount of code can be auto-generated in order to fully exploit the true benefits of Model-Driven Development. State-of-the-art tools such as IBM Rational Software Architect that let users define model transformations can be used by a development team to author transformations that not only automate code generation for software patterns but also incorporate a development team's best practices.

We automated only a subset of PoEAA patterns; a natural continuation of this work is to include more of Martin Fowler's patterns. In the previous section, we mentioned the limitations of our transformations. As future work, these limitations can be overcome. For the vision of MDD to become a reality, MDD proponents have to make an effort to write more transformations such as these. An online repository of transformations can be created. Such a repository can be used by MDD practitioners to share their work and to make use of other people's work.

References

[Bauer & King 2007] Christian Bauer and Gavin King, *Java Persistence with Hibernate*, Manning, 2007.

[Booch 2005] Grady Booch, *Object-Oriented Analysis and Design with Applications*, Addison-Wesley Professional, 2005.

[Czarnecki & Helsen 2003] Krzysztof Czarnecki and Simon Helsen, "Classification of Model Transformation Approaches," *in Proceedings of OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[Dae-Kyoo & Whittle 2005] Kim Dae-Kyo and Jon Whittle, "Generating UML Models from Domain Patterns," *In Proceedings of Third ACIS Int'l Conference on Software Engineering Research, Management and Applications*, 2005.

[Fowler 2002] Martin Fowler, *Patterns of Enterprise Application Architecture,* Addison-Wesley Professional, 2002.

[Fowler 2003] Martin Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, Addison-Wesley Professional, 2003.

[Gamma et al. 1995] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1995.

[Hibernate 2007] Hibernate Site, Online at www.hibernate.org, Accessed October, 2007.

[Kleppe et al 2003] Anneke Kleepe, Jos Warmer and Wim Bast, *MDA Explained*, Addison-Wesley Professional, 2003.

[Larman 2005] Craig Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, Prentice Hall PTR, 2005.

[Mellor & Scott 2004] Stephen J. Mellor, Kendall Scott, Axel UHL and Dirk Weise, *MDA Distilled: Principles of Model-Driven Architecture*, Addison-Wesley Professional, 2004.

[Nilsson 2006] Jimmy Nilsson, *Applying Domain-Driven Design and Patterns with Examples in C# and .NET*, Addison-Wesley Professional, 2006.

[OMG 2007] Object Management Group, Online at http://www.omg.org/, Accessed June, 2007.

[Pete & Heudecker 2006] Patrick Peak and Nick Heudecker, *Hibernate Quickly*, Manning, 2006.

[RAS 2007] Reusable Asset Specification home page, Online at http://www.omg.org/technology/documents/formal/ras.htm, Accessed October, 2007.

[Rumbaugh et al. 2005] James Rumbaugh, Ivar Jacobson, and Grady Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley Professional, 2005.

[Sendall & Kozakczynski 2003] Shane Sendall and Wojtek Kozaczynski, "Model transformation: The Heart and Soul of Model-Driven Software Development," IEEE Software, **20 (5)**, pp. 42-45, 2003.

[Selic 2006] Bran · Selic, "Model-Driven Development: its essence and opportunities," *in Proceedings of Ninth IEEE symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, 2006.

[Selic 2003] Bran Selic, "The Pragmatics of Model-Driven Development," IEEE Software, **20 (5)**, pp. 19-25, 2003.

[Shaw 1984] Mary Shaw, "Abstraction Techniques in Modern Programming Languages," IEEE Software, **1 (4)**, pp. 10-26, 1984.

[Weis et al 2003] Torben Weis, Andreas Ulbrich, and Kurt Geihs, "Model Metamorphosis," IEEE Software, **20 (5)**, pp. 46-51, 2003.

8 Appendix

8.1 Transformation source

8.1.1 DataMapper.javajet

```
<%@ jet package="com.ibm.xtools.transform.samples.modeltotext.jet"
class="DataMapperTemplate" imports="java.util.Map java.util.Iterator
com.ibm.xtools.transform.samples.modeltotext.JavaNameTool org.eclipse.uml2.uml.*
org.eclipse.emf.common.util.EList org.eclipse.emf.common.util.BasicEList java.util.List
utility.StringUtility java.lang.StringBuffer utility.StringUtility;"%>

<%
// retrieve the context
Map context = (Map)argument;
//extract the class from the context
org.eclipse.uml2.uml.Class mapperClass = (org.eclipse.uml2.uml.Class)context.get("class"

String pkgName =mapperClass.getPackage().getName();
String mapperClassName = mapperClass.getName();

String targetClassName  =
mapperClass.getName().substring(0,mapperClass.getName().lastIndexOf("Mapper"));
org.eclipse.uml2.uml.Class tdgClass  =
(org.eclipse.uml2.uml.Class)mapperClass.getModel().getNestedPackage("datasource").getMem
r(targetClassName+"TDG");
String tdgClassName = targetClassName+"TDG";

Operation insertOperation = (Operation) tdgClass.getOwnedOperation("insert",null,null);
Operation updateOperation = (Operation) tdgClass.getOwnedOperation("update",null,null);
Operation deleteOperation = (Operation) tdgClass.getOwnedOperation("delete",null,null);

EList parameterListInsertOperation = insertOperation.getOwnedParameters();
Iterator parameterListInsertOperationIterator = parameterListInsertOperation.iterator();

EList parameterListUpdateOperation = updateOperation.getOwnedParameters();
Iterator parameterListUpdateOperationIterator = parameterListUpdateOperation.iterator();

EList parameterListDeleteOperation = deleteOperation.getOwnedParameters();
Iterator parameterListDeleteOperationIterator = parameterListDeleteOperation.iterator();


StringBuffer operationString = new StringBuffer();
EList mapperClassOperationList = mapperClass.getAllOperations();
Iterator mapperClassOperationListIterator = mapperClassOperationList.iterator();

%>
<%="package "+ pkgName+";"%>


<%="import java.util.ArrayList;"%>
<%="import java.util.List;"%>
<%="import java.sql.SQLException;"%>
<%="import java.sql.ResultSet;"%>
<%="import org.dsrg.UoW.ObjectRemovedException;"%>
<%="import org.dsrg.UoW.UoW;"%>
<%="import org.dsrg.dom.DomainObjectCreationException;"%>
<%="import java.sql.SQLException;"%>
<%="import org.dsrg.dom.GenericMapper;"%>
<%="import org.dsrg.dom.MapperException;"%>
<%="import domain."+targetClassName+";"%>
<%="import datasource."+targetClassName+"Finder;"%>
<%="import datasource."+targetClassName+"TDG;"%>
<%="import org.dsrg.dom.MetaDomainObject;"%>



<%="public class " + mapperClassName+ "  implements GenericMapper<"+targetClassName+">{"

<%
   while(mapperClassOperationListIterator.hasNext()){
      Operation operation = (Operation)mapperClassOperationListIterator.next();
```

```
      if(operation.getName().equals("insert")){
           operationString = new StringBuffer();
           operationString.append("\n\tpublic void insert("+targetClassName+"
"+targetClassName.toLowerCase()+")throws MapperException{");
           operationString.append("\n\t\t try{");
           operationString.append("\n\t\t\t"+tdgClassName+".insert(");
           StringBuffer parameters = new StringBuffer();
           while(parameterListInsertOperationIterator.hasNext()){
           Parameter parameter = (Parameter)parameterListInsertOperationIterator.next();

parameters.append(targetClassName.toLowerCase()+".get"+StringUtility.upperFirstCharacter
arameter.getName())+"(),");
           }

operationString.append(StringUtility.removeLastCharacter(parameters.toString()));
           operationString.append(");");
           operationString.append("\n\t\t}catch(SQLException e){");
           operationString.append("\n\t\t\t throw new MapperException(e);");
           operationString.append("\n\t\t}");
           operationString.append("\n\t}");%>
           <%=operationString.toString()%>
<%    }//end of insert

      if(operation.getName().equals("update")){
          operationString = new StringBuffer();
           operationString.append("\n\tpublic void update("+targetClassName+"
"+targetClassName.toLowerCase()+")throws MapperException{");
           operationString.append("\n\t\t       try{");
           operationString.append("\n\t\t\t"+tdgClassName+".update(");
           StringBuffer parameters = new StringBuffer();
           while(parameterListUpdateOperationIterator.hasNext()){
           Parameter parameter = (Parameter)parameterListUpdateOperationIterator.next();

parameters.append(targetClassName.toLowerCase()+".get"+StringUtility.upperFirstCharacter
arameter.getName())+"(),");
           }

operationString.append(StringUtility.removeLastCharacter(parameters.toString()));
           operationString.append(");");
           operationString.append("\n\t\t}catch(SQLException e){");
           operationString.append("\n\t\t\t throw new MapperException(e);");
           operationString.append("\n\t\t}");
           operationString.append("\n\t}"); %>
           <%=operationString.toString()%>
<%    } // end of update

      if(operation.getName().equals("delete")){
          operationString = new StringBuffer();
          operationString.append("\n\t public void delete("+targetClassName+"
"+targetClassName.toLowerCase()+")throws MapperException{");
          operationString.append("\n\t\t try{");

          operationString.append("\n\t\t\t"+tdgClassName+".delete(");
          StringBuffer parameters = new StringBuffer();

          while(parameterListDeleteOperationIterator.hasNext()){
           Parameter parameter = (Parameter)parameterListDeleteOperationIterator.next();

parameters.append(targetClassName.toLowerCase()+".get"+StringUtility.upperFirstCharacter
arameter.getName())+"(),");
          }

          operationString.append(StringUtility.removeLastCharacter(parameters.toString())
          operationString.append(");");
          operationString.append("\n\t\t}catch(SQLException e){");
          operationString.append("\n\t\t\t throw new MapperException(e);");
          operationString.append("\n\t\t}");
          operationString.append("\n\t}");%>

           <%=operationString.toString()%>
<%    } // end of delete

if(operation.getName().equals("map")){
          operationString = new StringBuffer();
          operationString.append("\n\t public static "+targetClassName+" map(long id)
throws SQLException, DomainObjectCreationException{");
          operationString.append("\n\t\t try{");
          operationString.append("\n\t\t\t return mapKnown(id);");
          operationString.append("\n\t\t}catch(DomainObjectCreationException e){");
          operationString.append("\n\t\t\t System.out.println(e.getMessage());");
          operationString.append("\n\t\t} catch(ObjectRemovedException e){");
```

```
          operationString.append("\n\t\t\t System.out.println(e.getMessage());");
          operationString.append("\n\t\t}");
          operationString.append("\n\t\t ResultSet rs
="+targetClassName+"Finder.find(id);");
          operationString.append("\n\t\t if(!rs.next()) throw new
DomainObjectCreationException(\"Does not exist\");");
          operationString.append("\n\t\t return get"+targetClassName+"(rs);");
          operationString.append("\n\t}");%>
           <%=operationString.toString()%>

<%    } //end of map

          if(operation.getName().equals("get"+targetClassName+"s")){
          operationString = new StringBuffer();
          operationString.append("\n\t public static List<"+targetClassName+">
"+operation.getName()+"() throws SQLException{");
          operationString.append("\n\t\t ResultSet rs =
"+targetClassName+"Finder.findAll();");
          operationString.append("\n\t\tList<"+targetClassName+"> list = new
ArrayList<"+targetClassName+">();");
          operationString.append("\n\t\t while (rs.next()){");
          operationString.append("\n\t\t\t list.add(get"+targetClassName+"(rs));");
          operationString.append("\n\t\t}");
          operationString.append("\n\t\treturn list;");
          operationString.append("\n\t}");
          %>

           <%=operationString.toString()%>
<%    } //end of getAll

          if(operation.getName().equals("mapKnown")){
          operationString = new StringBuffer();
          operationString.append("\n\t public static "+targetClassName+"
"+operation.getName()+"(long id) throws
DomainObjectCreationException,ObjectRemovedException{");
          operationString.append("\n\t\t MetaDomainObject<"+targetClassName+"> mdo = new
MetaDomainObject<"+targetClassName+">("+targetClassName+".class,id);");
          operationString.append("\n\t\tif(UoW.getCurrent().hasObject(mdo)){");
          operationString.append("\n\t\t\t return ("+targetClassName+")
UoW.getCurrent().getObject(mdo);");
          operationString.append("\n\t\t}");
          operationString.append("\n\t\t throw new DomainObjectCreationException(\"Does r
exist\");");
          operationString.append("\n\t}");
          %>
           <%=operationString.toString()%>
<%    } //end of mapKnown %>
<%} //end of while%>

<%
          StringBuffer buffer = new StringBuffer();
          buffer.append("\n\t private static "+targetClassName+"
get"+targetClassName+"(ResultSet rs)throws SQLException{");
          buffer.append("\n\t\t "+targetClassName+" "+targetClassName.toLowerCase()+" = r
"+targetClassName+"(rs.getLong(\"id\"),");
          buffer.append("rs.getLong(\"version\"),");

          org.eclipse.uml2.uml.Class targetClass = (org.eclipse.uml2.uml.Class)
mapperClass.getPackage().getMember(targetClassName);
          EList attributeList = targetClass.getOwnedAttributes();
          Iterator attributeListIterator = attributeList.iterator();
          StringBuffer attributeListBuffer = new StringBuffer();
          while(attributeListIterator.hasNext()){
                    Property property = (Property) attributeListIterator.next();

   attributeListBuffer.append("rs.get"+StringUtility.upperFirstCharacter(property.getTyp
).getName())+"(\""+property.getName()+"\"),");
          }
          buffer.append(StringUtility.removeLastCharacter(attributeListBuffer.toString())
          buffer.append(");");
          buffer.append("\n\t\t
UoW.getCurrent().registerClean("+targetClassName.toLowerCase()+");");
          buffer.append("\n\t\t return "+targetClassName.toLowerCase()+";");
          buffer.append("\n\t}");%>

           <%=buffer.toString()%>


<%="}"%>
```

8.1.2 TableDataGateway.javajet

```
<%@ jet class="TableDataGatewayTemplate"
package="com.ibm.xtools.transform.samples.modeltotext.jet"  imports="java.util.Map
java.util.Iterator com.ibm.xtools.transform.samples.modeltotext.JavaNameTool
org.eclipse.uml2.uml.* java.util.List utility.StringUtility java.lang.StringBuffer
org.eclipse.emf.common.util.EList;" %>

<%
// retrieve the context
Map context = (Map)argument;

//extract the class from the context
org.eclipse.uml2.uml.Class TDGclass = (org.eclipse.uml2.uml.Class)context.get("class");
String pkgName =TDGclass.getPackage().getName();
String className= TDGclass.getName();
StringBuffer parameterBuffer;

StringBuffer tableNameString = new StringBuffer();
StringBuffer SQLstmtString = new StringBuffer();

StringBuffer operationString=null;
EList parameterList;
Iterator parameterListIterator;

EList attributeList = TDGclass.getOwnedAttributes();
Iterator attributeListItarator = attributeList.iterator();

EList operationList = TDGclass.getOwnedOperations();
Iterator operationListIterator = operationList.iterator();

%>

<%="package"+" "+ pkgName +";" %>

<%="import java.sql.PreparedStatement;"%>
<%="import java.sql.ResultSet;"%>
<%="import java.sql.SQLException;"%>
<%="import java.sql.Connection;"%>

<%="import org.dsrg.ts.UniqueIdFactory;"%>
<%="import org.dsrg.ts.threadLocal.DbRegistry;"%>

<%="public class "+ className + "{" %>

 <% while(attributeListItarator.hasNext()){
                    Property attribute = (Property)attributeListItarator.next();
                    if(attribute.getName().equalsIgnoreCase("TABLE_NAME")){
                    tableNameString.append("\n\t private static final"+"
"+attribute.getType().getName()+" "+attribute.getName()+" =
"+attribute.getDefault()+";"+"\n");
          }else{
                    SQLstmtString.append("\t private static final"+"
"+attribute.getType().getName()+" "+attribute.getName()+" =
"+attribute.getDefault()+";"+"\n");
          }
   }
%>
  <%=tableNameString%>
  <%=SQLstmtString.toString()%>

   <% while(operationListIterator.hasNext()){

   Operation operation =(Operation) operationListIterator.next();

   if(operation.getName().equalsIgnoreCase("insert")){
          operationString = new StringBuffer();
          operationString.append("\n\t public static int "+operation.getName()+"(");

          parameterList = operation.getOwnedParameters();
          parameterListIterator = parameterList.iterator();
          parameterBuffer = new StringBuffer();

          while(parameterListIterator.hasNext()){
```

```
                    Parameter parameter = (Parameter)parameterListIterator.next();
                    parameterBuffer.append(parameter.getType().getName()+"
"+parameter.getName()+",");
          }


   operationString.append(StringUtility.removeLastCharacter(parameterBuffer.toString()))
          operationString.append(") throws SQLException {");
          operationString.append("\n");
          operationString.append("\n\t\t Connection con =DbRegistry.getDbConnection();");
          operationString.append("\n\t\t PreparedStatement ps =
con.prepareStatement(INSERT_STRING);");
          int i = 1;
          parameterListIterator = parameterList.iterator();

          while(parameterListIterator.hasNext()){
                    Parameter parameter = (Parameter)parameterListIterator.next();
                    operationString.append("\n\t\t ps.set");


   if(parameter.getType().getName().equalsIgnoreCase("Integer")||parameter.getType().getI
me().equalsIgnoreCase("Int")){
                              operationString.append("Int("+i+","+parameter.getName()+");'
                    }
                    else if(parameter.getType().getName().equalsIgnoreCase("Boolean")){
                    operationString.append("Boolean("+i+","+parameter.getName()+");");
                    }
                    else if(parameter.getType().getName().equalsIgnoreCase("Byte")){
                    operationString.append("Byte("+i+","+parameter.getName()+");");
                    }
                    else if(parameter.getType().getName().equalsIgnoreCase("Double")){
                    operationString.append("Double("+i+","+parameter.getName()+");");
                    }
                    else if(parameter.getType().getName().equalsIgnoreCase("Float")){
                    operationString.append("Float("+i+","+parameter.getName()+");");
                    }
                    else if(parameter.getType().getName().equalsIgnoreCase("Long")){
                    operationString.append("Long("+i+","+parameter.getName()+");");
                    }
                    else if(parameter.getType().getName().equalsIgnoreCase("Short")){
                    operationString.append("Short("+i+","+parameter.getName()+");");
                    }
                    else if(parameter.getType().getName().equalsIgnoreCase("String")){
                    operationString.append("String("+i+","+parameter.getName()+");");
                    }
                    else {

   operationString.append(StringUtility.upperFirstCharacter(parameter.getType().getName(
+"("+i+","+parameter.getName()+");");
                    }
                    i++;
          }
          operationString.append("\n\t\t return ps.executeUpdate();");
          operationString.append("\n\t}");%>
   <%=operationString.toString()%>
<% } //end of insert
 if(operation.getName().equalsIgnoreCase("update")){

      operationString = new StringBuffer();
      operationString.append("\n\t public static int "+operation.getName()+"(");

      parameterList = operation.getOwnedParameters();
      parameterListIterator = parameterList.iterator();
      parameterBuffer = new StringBuffer();

      while(parameterListIterator.hasNext()){
          Parameter parameter = (Parameter)parameterListIterator.next();
          parameterBuffer.append(parameter.getType().getName()+"
"+parameter.getName()+",");
      }

   operationString.append(StringUtility.removeLastCharacter(parameterBuffer.toString()))
      operationString.append(") throws SQLException {");
      operationString.append("\n");
      operationString.append("\n\t\t Connection con =DbRegistry.getDbConnection();");
      operationString.append("\n\t\t PreparedStatement ps =
con.prepareStatement(UPDATE_STRING);");
      int i = 1;
      parameterListIterator = parameterList.iterator();
      while(parameterListIterator.hasNext()){
          Parameter parameter = (Parameter)parameterListIterator.next();
```

```
          operationString.append("\n\t\t ps.set");

                    if((parameter.getName().equalsIgnoreCase("id")== false &&
parameter.getType().getName().equalsIgnoreCase("Long")== false) &&
(parameter.getName().equalsIgnoreCase("version")== false &&
parameter.getType().getName().equalsIgnoreCase("Long") == false)){

   if(parameter.getType().getName().equalsIgnoreCase("Integer")||parameter.getType().getI
me().equalsIgnoreCase("Int")){
                              operationString.append("Int("+i+","+parameter.getName()+");'
                              }
                              else
if(parameter.getType().getName().equalsIgnoreCase("Boolean")){

   operationString.append("Boolean("+i+","+parameter.getName()+");");
                              }
                              else
if(parameter.getType().getName().equalsIgnoreCase("Byte")){

   operationString.append("Byte("+i+","+parameter.getName()+");");
                              }
                              else
if(parameter.getType().getName().equalsIgnoreCase("Double")){

   operationString.append("Double("+i+","+parameter.getName()+");");
                              }
                              else
if(parameter.getType().getName().equalsIgnoreCase("Float")){

   operationString.append("Float("+i+","+parameter.getName()+");");
                              }
                              else
if(parameter.getType().getName().equalsIgnoreCase("Long")){

   operationString.append("Long("+i+","+parameter.getName()+");");
                              }
                              else
if(parameter.getType().getName().equalsIgnoreCase("Short")){

   operationString.append("Short("+i+","+parameter.getName()+");");
                              }
                              else
if(parameter.getType().getName().equalsIgnoreCase("String")){

   operationString.append("String("+i+","+parameter.getName()+");");
                              }
                              else {

   operationString.append(StringUtility.upperFirstCharacter(parameter.getType().getName(
+"("+i+","+parameter.getName()+");");
                              }
                              i++;
                    }

           else if(parameter.getName().equalsIgnoreCase("id")== true &&
parameter.getType().getName().equalsIgnoreCase("Long")){
                              operationString.append("Long(");
                              operationString.append(parameterList.size()-1);
                              operationString.append(",id);");

                    }
          else if(parameter.getName().equalsIgnoreCase("version")&&
parameter.getType().getName().equalsIgnoreCase("Long")){
                              operationString.append("Long(");
                              operationString.append(parameterList.size());
                              operationString.append(",version);");
                    }

          }
          operationString.append("\n\t\t return ps.executeUpdate();");
          operationString.append("\n\t}");%>
   <%=operationString.toString()%>
<% }//end of update

   if(operation.getName().equalsIgnoreCase("delete")){

      operationString = new StringBuffer();
      operationString.append("\n\t public static int "+operation.getName()+"(");

      parameterList = operation.getOwnedParameters();
      parameterListIterator = parameterList.iterator();
```

```
      parameterBuffer = new StringBuffer();

      while(parameterListIterator.hasNext()){
          Parameter parameter = (Parameter)parameterListIterator.next();
          parameterBuffer.append(parameter.getType().getName()+"
"+parameter.getName()+",");
      }

   operationString.append(StringUtility.removeLastCharacter(parameterBuffer.toString()))
      operationString.append(") throws SQLException {");
      operationString.append("\n");
      operationString.append("\n\t\t Connection con =DbRegistry.getDbConnection();");
      operationString.append("\n\t\t PreparedStatement ps =
con.prepareStatement(DELETE_STRING);");
      int i = 1;
      parameterListIterator = parameterList.iterator();
      while(parameterListIterator.hasNext()){
          Parameter parameter = (Parameter)parameterListIterator.next();
          operationString.append("\n\t\t ps.set");

                    if((parameter.getName().equalsIgnoreCase("id")== false &&
parameter.getType().getName().equalsIgnoreCase("Long")== false) &&
(parameter.getName().equalsIgnoreCase("version")== false &&
parameter.getType().getName().equalsIgnoreCase("Long") == false)){

   if(parameter.getType().getName().equalsIgnoreCase("Integer")||parameter.getType().getI
me().equalsIgnoreCase("Int")){
                              operationString.append("Int("+i+","+parameter.getName()+");'
                              }
                              else
if(parameter.getType().getName().equalsIgnoreCase("Boolean")){

   operationString.append("Boolean("+i+","+parameter.getName()+");");
                              }
                              else
if(parameter.getType().getName().equalsIgnoreCase("Byte")){

   operationString.append("Byte("+i+","+parameter.getName()+");");
                              }
                              else
if(parameter.getType().getName().equalsIgnoreCase("Double")){

   operationString.append("Double("+i+","+parameter.getName()+");");
                              }
                              else
if(parameter.getType().getName().equalsIgnoreCase("Float")){

   operationString.append("Float("+i+","+parameter.getName()+");");
                              }
                              else
if(parameter.getType().getName().equalsIgnoreCase("Long")){

   operationString.append("Long("+i+","+parameter.getName()+");");
                              }
                              else
if(parameter.getType().getName().equalsIgnoreCase("Short")){

   operationString.append("Short("+i+","+parameter.getName()+");");
                              }
                              else
if(parameter.getType().getName().equalsIgnoreCase("String")){

   operationString.append("String("+i+","+parameter.getName()+");");
                              }
                              else {

   operationString.append(StringUtility.upperFirstCharacter(parameter.getType().getName(
+"("+i+","+parameter.getName()+");");
                              }
                              i++;
                    }

           else if(parameter.getName().equalsIgnoreCase("id")== true &&
parameter.getType().getName().equalsIgnoreCase("Long")){
                              operationString.append("Long(");
                              operationString.append(parameterList.size()-1);
                              operationString.append(",id);");

                    }
          else if(parameter.getName().equalsIgnoreCase("version")&&
parameter.getType().getName().equalsIgnoreCase("Long")){
```

```
                        operationString.append("Long(");
                        operationString.append(parameterList.size());
                        operationString.append(",version);");
                    }

            }
            operationString.append("\n\t\t return ps.executeUpdate();");
            operationString.append("\n\t}");%>
        <%=operationString.toString()%>
    <%}//end of update%>

<%} //end of while%>

<%="}"%>
```

8.1.3 Finder.javajet

```
<%@ jet package="com.ibm.xtools.transform.samples.modeltotext.jet" class="FinderTemplate
imports="java.util.Map java.util.Iterator
com.ibm.xtools.transform.samples.modeltotext.JavaNameTool org.eclipse.uml2.uml.*
org.eclipse.emf.common.util.EList org.eclipse.emf.common.util.BasicEList java.util.List
utility.StringUtility java.lang.StringBuffer utility.StringUtility;"%>


<%
// retrieve the context
Map context = (Map)argument;

//extract the class from the context
org.eclipse.uml2.uml.Class finderClass = (org.eclipse.uml2.uml.Class)context.get("class"

String pkgName = finderClass.getPackage().getName();
String className= finderClass.getName();

EList attributeList = finderClass.getOwnedAttributes();
Iterator attributeListIterator = attributeList.iterator();

EList operationList = finderClass.getOwnedOperations();
Iterator operationListIterator = operationList.iterator();

StringBuffer propertyString;
StringBuffer operationString;



%>
package <%=pkgName%>;


<%="import java.sql.Connection;"%>
<%="import java.sql.PreparedStatement;"%>
<%="import java.sql.ResultSet;"%>
<%="import java.sql.SQLException;"%>
<%="import org.dsrg.ts.threadLocal.DbRegistry;"%>


<%="public class"%> <%=className%> <%="{"%>

<%
   while(attributeListIterator.hasNext()){

   Property property = (Property) attributeListIterator.next();
   propertyString = new StringBuffer();

   propertyString.append("\n\t");

   if(property.getVisibility()!=null){
   propertyString.append(property.getVisibility().getName()+" ");
   }

   if(property.isLeaf()== true){
      propertyString.append(" final ");
    }

    if(property.isStatic()== true){
      propertyString.append(" static ");
    }
```

```
    if(property.getType()!= null){
       propertyString.append(property.getType().getName());
    }

    propertyString.append(" ");
    propertyString.append(property.getName());

   if(property.getDefault() != null){
    propertyString.append(" = "+property.getDefault());
   }
    propertyString.append(";"); %>
   <%=propertyString.toString()%>
<%}%>

<%
  while(operationListIterator.hasNext()){

  Operation operation = (Operation) operationListIterator.next();
  operationString = new StringBuffer();

   if(operation.getName().equalsIgnoreCase("find")){
         operationString.append("\n\tpublic static ResultSet find(long id) throws
SQLException{");
         operationString.append("\n\t\tConnection con = DbRegistry.getDbConnection();");
         operationString.append("\n\t\tPreparedStatement ps =
con.prepareStatement(SELECT_STRING);");
         operationString.append("\n\t\tps.setLong(1, id);");
         operationString.append("\n\t\treturn ps.executeQuery();");
         operationString.append("\n\t}");
   }

   else if(operation.getName().equalsIgnoreCase("findAll")){
         operationString.append("\n\tpublic static ResultSet findAll() throws
SQLException{");
         operationString.append("\n\t\tConnection con = DbRegistry.getDbConnection();");
         operationString.append("\n\t\tPreparedStatement ps =
con.prepareStatement(SELECT_ALL_STRING);");
         operationString.append("\n\t\treturn ps.executeQuery();");
         operationString.append("\n\t}");

   }

   else{
         if(operation.getVisibility()!= null){
         operationString.append("\n\t"+operation.getVisibility().getName()+" ");
         }

      if(operation.isLeaf()==true){
         operationString.append(" final ");
         }

      if(operation.isStatic()==true){
         operationString.append(" static ");
         }

         if(operation.getReturnResult()!= null ){
         operationString.append(operation.getReturnResult().getType().getName()+" ");
         }
         operationString.append(operation.getName()+"(){");
         operationString.append("\n\n\n");
         operationString.append("\n\t}");
   }


  %>


   <%=operationString.toString()%>

<%}%>
<%="}"%>
```

8.1.4 LazyLoad.javajet

```
<%@ jet class="IdentityMapTemplate"
package="com.ibm.xtools.transform.samples.modeltotext.jet"  imports="java.util.Map
java.util.List java.util.Iterator
com.ibm.xtools.transform.samples.modeltotext.JavaNameTool utility.StringUtility
```

```
org.eclipse.uml2.uml.*" %>
<%
// retrieve the context
Map context = (Map)argument;

// extract things from the context
org.eclipse.uml2.uml.Class targetClass = (org.eclipse.uml2.uml.Class)context.get("class"
String className= targetClass.getName();
String pkgName = targetClass.getPackage().getName();
String domainClassName =
targetClass.getName().substring(0,targetClass.getName().lastIndexOf("Proxy"));
%>

package <$= pkgName $>;


public class <%=className%> implements <%="I"+domainClassName%> {

    private <%=domainClassName%> inner<%=domainClassName%> = null;
   private long id = 0;

   public <%=className%>(long id) {
      this.id = id;
   }

   private synchronized <%=domainClassName%> getInner<%=domainClassName%>() {
      if(inner<%=domainClassName%>==null)
          try {
                    inner<%=domainClassName%> = <%=domainClassName%>Mapper.map(id);
          } catch (Exception e) {
                    throw new ProxyException(e);
          }
      return inner<%=domainClassName%>;
   }

<%
 List operationList = targetClass.getOwnedOperations();
 Iterator operationListIterator = operationList.iterator();
 StringBuffer buffer = null;

 while(operationListIterator.hasNext()){
   Operation operation = (Operation)operationListIterator.next();
   buffer.append("\n\t\t"+operation.getVisibility().getName()+"
"+operation.getReturnResult().getType().getName()+" "+operation.getName()+"(){");
      if(operation.getReturnResult()!= null ){
          buffer.append("\n\t\t\t return"+"
getInner"+domainClassName+"()."+operation.getName()+"();");
          buffer.append("\n\t\t }");
      }
      else{
          buffer.append("\n\t\t\t
"+"getInner"+domainClassName+"()."+operation.getName()+"(");
          List parameterList =operation.getOwnedParameters();
          Iterator parameterListIterator = parameterList.iterator();
          StringBuffer paraBuffer = new StringBuffer();
          StringBuffer argBuffer = new StringBuffer();
           while(parameterListIterator.hasNext()){
            Parameter parameter = (Parameter) parameterListIterator.next();
            paraBuffer.append(parameter.getType().getName()+" "+parameter.getName()+",");
            argBuffer.append(parameter.getName()+",");
           }
          buffer.append(StringUtility.removeLastCharacter(paraBuffer.toString()));
          buffer.append("){");
          buffer.append("\n\t\t\t getInner"+domainClassName+"()."+operation.getName()+"("  .
          buffer.append(StringUtility.removeLastCharacter(argBuffer.toString()));
          buffer.append(");");
          buffer.append("\n\t\t}");
      }
 }

%>
   <%=buffer.toString()%>
}
```