

Model Driven Development: A Comprehensive Case Study

Rajiv Abraham

**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**

August 2007

© Rajiv Abraham, 2007



Library and
Archives Canada

Bibliothèque et
Archives Canada

Published Heritage
Branch

Direction du
Patrimoine de l'édition

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 978-0-494-34612-9

Our file Notre référence

ISBN: 978-0-494-34612-9

NOTICE:

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

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

AVIS:

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

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

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

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

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

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


Canada

ABSTRACT

Model Driven Development: A Comprehensive Case Study

Rajiv Abraham

The major complexity in creating software systems lies in the understanding of the problem domain for which the software is developed. Models help analyzing the problem domain more effectively as they provide a higher level of abstraction by filtering out the low-level details. Model Driven Development is a methodology that advocates the use of models as the primary artifacts that drive the development of software instead of serving as informal sketches. In this thesis, we demonstrate the application of MDD to the development of a non-trivial software system using a state of the art tool: IBM's Rational Software Architect (RSA). We also investigate how MDD attempts to solve the problem of model-code synchronization through Round-Trip Engineering. With the help of this case study, we present the limitations and guidelines learned regarding the use of the state of the art for MDD, highlighting the difference between theory and practice.

Acknowledgments

I would like to extend my gratitude first and foremost to my supervisor Professor Patrice Chalin. The guidance provided by him throughout the thesis has been invaluable. Every meeting is an opportunity to learn. He has been a role model both professionally and personally.

Secondly, I would like to thank the DSRG group for their combined effort in reviewing my thesis. It is a privilege to be a member of this group. I would especially like to thank Stephen Barrett, Perry James and Daniel Sinnig for their detailed review of the final versions of this thesis.

Last but not the least, I would like to thank my family and my fiance Manali, for their support throughout the thesis.

Table of Contents

List of Figures	vii
List of Tables.....	ix
List of Acronyms.....	x
1 Introduction	1
1.1 Importance of Models	1
1.2 Problems	2
1.3 Contributions.....	3
1.4 Outline	4
2 Background and Related Work	5
2.1 Unified Modeling Language (UML).....	5
2.1.1 UML and Modeling.....	5
2.1.2 Properties.....	7
2.1.3 Operations	8
2.1.4 Constraints.....	9
2.1.5 Stereotypes	9
2.2 Model Driven Development (MDD).....	9
2.2.1 The Need for Models.....	9
2.2.2 Essentials Components of MDD	10
2.2.3 Model Driven Architecture (MDA).....	11
2.2.4 MDD Vision.....	12
2.3 Test Driven Development (TDD)	12
2.3.1 Testing Frameworks	13
2.3.2 Methodology	13
2.3.3 Advantages and Disadvantages	14
2.4 Refactoring.....	14
2.5 Related Work	18
3 Rational Software Architect (RSA).....	19
3.1 Scope of Usage of RSA	19
3.2 Key RSA Features for MDD.....	19
4 Case Study: Development of a Project Management System (PMS).....	24
4.1 Iteration Presentation Style	24
4.2 Iteration 0: Preliminary Analysis and Creation of a Domain Model.....	24
4.3 Iteration 1: MDD for Basic Features.....	27
4.3.1 Domain Model: Order of Tasks and Folders	28
4.3.2 Basic Features for Tasks and Folders.....	32
4.4 Iteration 2: Preliminary Thick Client UI	36
4.4.1 Eclipse Rich Client Platform (RCP).....	36
4.4.2 Feature: Basic UI features for Tasks	38
4.4.3 Feature: Basic UI features for Folders.....	40
4.5 Iteration 3: View All Tasks for a Folder	41
4.5.1 Feature: View All Tasks of Folder	41
4.5.2 Feature: View All Tasks by Folder (UI).....	42
4.5.3 Rework on Remove Task	43
4.6 Iteration 4: TDD for Sorting Tasks	43
4.6.1 Design Rationale	43
4.6.2 JUnit Tests for Sorting	44

4.6.3	Sorting in UI.....	50
4.6.4	Lessons Learned.....	50
4.7	Iteration 5: Reordering of Tasks	51
4.7.1	Initial UI investigation.....	51
4.7.2	MDD: Unique Task ID.....	54
4.7.3	MDD: RankExpert for Reordering Tasks.....	54
4.7.4	MDD: Reordering Tasks	56
4.7.5	Providing the Right Tasks to the RankExpert	57
4.8	Iteration 6: Share Tasks.....	62
4.8.1	Domain Analysis	62
4.8.2	Feature: Sharing Tasks	63
4.8.3	Sharing Tasks in UI.....	64
4.8.4	Refactoring.....	65
4.8.5	Class Diagram of PMS.....	66
4.9	Iteration 7: Persistence	67
4.9.1	Introduction	67
4.9.2	Database Table Schema.....	67
4.9.3	ID Allocation to Domain Entities.....	68
4.9.4	Unit of Work	68
4.9.5	Identity Map	70
4.9.6	Data Mapper and Table Data Gateway.....	71
4.9.7	Virtual Proxy	72
4.10	Case Study Summary	74
5	RSA Limitations and Guidelines.....	81
5.1	Reverse Transformation Done For Entire Java Project.....	81
5.2	Synchronization Issues in Reverse Transformation	83
5.2.1	Limitation: No Link Created by Reverse Transformation for Added Code.	83
5.2.2	Importance of Reverse Transformation for the Adoption of MDD.....	83
5.2.3	Workarounds for Reverse Transformations in RSA.....	84
5.3	Reverse Transformation of UML Profiles	87
5.4	UML-to-Java Transformation Impacts.....	89
5.5	Sequence Diagram Support.....	90
5.6	Java Specific Limitations	91
5.6.1	Accessor and Mutators	91
5.6.2	Lack of Complete Support for Java Generics.....	91
5.6.3	Method Parameters and Return Values	92
5.6.4	Lack of Support for Enumeration Operations	93
5.6.5	Final Keyword for Method Parameters	94
5.7	General Guidelines.....	94
5.7.1	Model Public Interfaces First	94
5.7.2	Detecting Erasure of Code	95
5.7.3	Undo Forward Transformation Effect	96
6	Conclusions and Future Work.....	97
	References	100
7	Appendix	104
7.1	RSA.....	104
7.1.1	Importing Java Model Libraries	104
7.1.2	View Enumeration Compartments	104
7.2	Eclipse.....	104

List of Figures

Figure 1: Types of UML Diagrams [Fowler 2003].....	6
Figure 2: Class Diagram explaining basic UML concepts.....	7
Figure 3: MDA software development life cycle [Kleppe et al. 2003].....	11
Figure 4: Example of Code that has a “bad smell”	16
Figure 5: Example of Code after Refactoring	16
Figure 6: Modeling Perspective in RSA	20
Figure 7: The mechanism used by RSA to maintain a link between model and code	21
Figure 8: Snapshot of generated code with @generated tag and inline comments.....	22
Figure 9: Model Merging Window during Reverse Transformation	23
Figure 10: Initial Domain Model	26
Figure 11: Initial class diagram.....	28
Figure 12: Snapshot of properties for <code>Folder.tasks</code>	29
Figure 13: Collection tab in transformation configuration file	29
Figure 14: RSA support for Design Patterns	31
Figure 15: Singleton code automatically generated by RSA	32
Figure 16: Class Diagram for Basic Features for Task and Folder.....	34
Figure 17: Empty Method body for <code>Folder.newFolder()</code> generated by RSA after Forward Transformation.....	35
Figure 18: Programmer Filled Method body for <code>Folder.newFolder()</code> after Forward Transformation.....	35
Figure 19: Eclipse Platform Architecture	37
Figure 20: Class diagram for <code>FolderTaskListUIWrapper</code> and <code>ITaskListViewer</code>	39
Figure 21: Table View of Tasks.....	39
Figure 22: Tree view of folders	40
Figure 23: Class diagram for <code>FolderUIWrapper</code>	40
Figure 24: Context menu for Add Folder and Delete Folder	41
Figure 25: Class diagram after implementing View All Tasks feature.....	42
Figure 26: View Tasks by Folder.....	42
Figure 27: Code showing <code>InverseImportanceLevelComparator</code> undefined while writing first test.....	45
Figure 28: Shows Eclipse Quick Fix feature	45
Figure 29: Shows Class <code>ImportanceLevelComparator</code> created by Eclipse with default return value of zero	46
Figure 30: compare method for class <code>ImportanceLevelComparator</code>	46
Figure 31: JUnit Eclipse View showing all tests pass indicated by the green bar	46
Figure 32: Code for <code>TestCompareTasksByInverseImportanceLevel</code>	48
Figure 33: Code for <code>TestTaskSortByImportanceLevel</code>	49
Figure 34: Sorting tasks by column widget	50
Figure 35: Illustration of task reordering through the UI’s drag-and-drop capability: moving Task 1 to after Task 3	51
Figure 36: Class Diagram showing UI implementation for DnD feature	53
Figure 37: Class <code>Task</code> with UUID	54
Figure 38: Illustration shows example of class <code>Folder</code> not being a candidate for reordering tasks. <code>Folder A</code> cannot place <code>Task 3</code> before or after <code>Task 6</code>	55
Figure 39: Sequence Diagram showing <code>TaskTreeDropAdapter</code> checking drop location	59
Figure 40: Sequence Diagram showing <code>TaskTreeDropAdapter</code> reordering with next task	59

Figure 41: Class Diagram for Reordering Tasks feature	60
Figure 42: Task View showing Drag and Drop feature	61
Figure 43: Class Diagram showing Domain Layer after implementing reorder feature.....	61
Figure 44: Why a folder cannot remove a subfolder's tasks.....	62
Figure 45: Class Diagram after adding Share Task feature	64
Figure 46: Snapshot of UI showing a task dragged and dropped for sharing between folders.....	65
Figure 47: Snapshot of Context Menu for Removing shared Tasks	65
Figure 48: Class Diagram of FolderView before refactoring	66
Figure 49: Class Diagram of FolderView and ContextMenu after <i>Extract Class</i>	66
Figure 50: Class Diagram for Presentation Layer.....	66
Figure 51: Class Diagram of Domain Layer.....	67
Figure 52: Database Table Schema for Folder and Task Entities	68
Figure 53: Class Diagram for ID generation.....	68
Figure 54: Class Diagram showing functionality for Batch Updates	70
Figure 55: Class Diagram showing functionality for maintaining unique domain entity instances	71
Figure 56: Sequence Diagram showing behaviour of finding a task in TaskMapper	71
Figure 57: Class Diagram showing functionality for mapping objects to persistence.....	72
Figure 58: Class Diagram showing functionality to prevent deadlock for loading domain entities	73
Figure 59: Sequence Diagram showing behavior of loading a task from the database	74
Figure 60: Package Diagram for the Project Management System	77
Figure 61: Class Diagram of the Presentation Layer	78
Figure 62: Class Diagram showing Domain and Technical Services Layers	79
Figure 63: Class Diagram showing Unit Tests for the Domain Layer.....	80
Figure 64: Conflict Merge during Reverse Transformation from code to model	82
Figure 65: Eclipse Content Assist for Tag Template.....	86
Figure 66: Singleton Profile in RSA	87
Figure 67: Visualization of generated Java Class A	87
Figure 68: Model Merging Window during Reverse Transformation from Java to UML for the Singleton Pattern.....	88
Figure 69: Class Signature of RankComparator	91
Figure 70: Illustration showing a workaround for generics in method parameters	92
Figure 71: Public methods of RankExpert	95
Figure 72: Evolution of RankExpert during implementation.....	95
Figure 73: Code Style In Eclipse Preferences.....	105
Figure 74: Snapshot of Eclipse Code Templates	106
Figure 75: Comment Template for Methods.....	106
Figure 76: Comment Template to be pasted in the Pattern textbox for methods.....	107
Figure 77: Comment Template to be pasted in the Pattern textbox for getter methods.....	107
Figure 78: Comment Template to be pasted in the Pattern textbox for setter methods	107
Figure 79: Code Template to be pasted in the Pattern textbox for methods	107
Figure 80: Code Template to be pasted in the Methods Pattern textbox for throwing an Exception	107
Figure 81: Code Template to be pasted in the Pattern textbox for getter methods.....	107
Figure 82: Code Template to be pasted in the Pattern textbox for setter methods	108
Figure 83: Creating Templates in Eclipse.....	108

List of Tables

Table 1: Impact of UML-to-Java transformation on modified code	89
Table 2: Impact of UML-to-Java transformation on code if model is changed	90

List of Acronyms

Acronym	Definition
DnD	Drag-and-Drop, a UI capability.
IP	Intellectual Property
Java EE	Java Standard for building server side enterprise applications.
MDA	Model Driven Architecture: A framework for MDD
MDD	Model Driven Development: A methodology where models are primary artifacts of software development.
.NET	Framework for creating applications using Microsoft Technologies.
OCL	Object Constraint Language, declarative language for describing rules that apply to UML models
PIM	Platform Independent Model. A model abstraction used in Model Driven Architecture
PSM	Platform Specific Model: A model abstraction used in Model Driven Architecture
RSA	Rational Software Architect, a tool for MDD used in the case study.
UI	User Interface.
UML	Unified Modeling Language, a visual object modeling language

1 Introduction

1.1 Importance of Models

Software systems increase in complexity every decade. From the humble beginnings of solving simple mathematical problems, software now powers spaceships and complex medical instruments. Current practices in developing complex software involve advances in programming languages and IDEs (Integrated Development Environment). However, these advances have proven insufficient for managing the complexity of software development. This is because while these technical advances improve the implementation domain, the major complexity in creating software lies in understanding its domain of application, i.e., the problem domain. The semantic gap between the problem domain and the implementation domain is too wide for these technical advances to have a pronounced effect in understanding the problem domain [France & Rumpe 2007].

Therefore, there is a need to analyze the problem domain without being concerned with the implementation details [Hailpern & Tarr 2006]. This is normally achieved by creating abstractions of the problem domain. These abstractions or models are conveniently expressed using graphical notations. A pervasive graphical modeling notation is the Unified Modeling Language (UML) [UML 2007]. The UML defines a suite of graphical notations, based on a single meta-model, which helps in describing and defining software systems using the Object Oriented (OO) style [Fowler 2003].

Through models, developers can focus on the high-level design of a system instead of its low-level technical details. Using models, developers can communicate quickly and efficiently about design decisions [Larman 2004]. In addition, the models themselves serve as documentation for these high-level decisions, which can be very useful to maintenance developers who have to understand the system well after it is first deployed.

The desirability of such benefits has led to increasing amounts of research in the area of Model Driven Development (MDD), a software engineering approach that considers models to be the primary artifact of software development. Models as defined in MDD, have a precise syntax and semantics, and are machine-

readable. This forms the necessary basis of the MDD vision - the automatic generation of executable code from models with the help of tools.

1.2 Problems

Detailed case studies in MDD are quite rare. This is probably due in part to the fact that the industrial application of MDD is still in its infancy, so there are few cases to report. Other reasons could be that issues of Intellectual Property (IP) prevent those that have been carried out from being made public [France & Rumpe 2007] and that early adopters may be unwilling to face criticism. The case studies that have been published almost exclusively discuss only high-level architecture issues [Mattsson et al. 2007] or have a business focus [Guttman & Parodi 2007]. Another issue with current case studies is that they do not offer a realistic description of the *practice* of MDD. Simplified case studies [Quatrani & Palistrant 2006] or vendor white papers do not provide practical guidelines to developers interested in adopting MDD. Other case studies [Nilsson 2006] provide a realistic description of the practice of modeling but are not set in the context of MDD. Case studies also do not detail the role of tools in MDD. The successful adoption of MDD is dependent on the degree of automation provided by tools. Since MDD is a relatively new methodology, it is likely that tools work well only within certain limitations. Current literature does not provide adequate information about these limitations.

The biggest challenge in MDD is ensuring model-code synchronization [Haskins 1997]. Synchronization problems may occur due to the following reasons:

- The generated code may be optimized for better performance, making it hard to understand. Developers may change the code to make it easier to understand, using practices like refactoring of code.
- Many MDD tools provide only limited support for integrating generated code with human-written code, as the architectural choices made by the tools are hidden from the modeler and non-customizable [France & Rumpe 2007]. For this reason, once the initial code has been generated, the developers may simply ignore the tool and continue development directly on the code itself.

Reverse engineering the changes made in the code back to the model is complex. This is because, for automatic conversion from code to model, current tools cannot perform the same level of abstraction as done by humans [Selic 2003]. If the changes made in the code cannot be propagated to the model completely then the model will represent an incomplete and incorrect picture of the software. Such an incomplete model can only be partially reused across similar applications. Hence, model-code synchronization is important to facilitate maximum model reuse.

In addition, there has been an increase in the adoption of Agile methodologies in software development [Ambler 2007]. Test Driven Development (TDD) [Astels 2003] is one of the popular Agile methodologies [Saiedian & Janzen 2005]. TDD advocates writing tests before implementation. All production code must have associated tests. Agile MDD [Ambler 2004], an agile activity related to MDD, advises the combination of MDD and TDD. However, there is a lack of detailed examples describing the application of such a combination.

1.3 Contributions

The main contributions of this thesis address the previously mentioned problems through

- A *detailed case study* that applies MDD to the creation of a software application of *non-trivial size*, namely, a Project Management System (PMS).
- A special focus during the case study execution on how MDD attempts to solve the problem of *model-code synchronization* using a state-of-the-art tool, namely **IBM Rational Software Architect (RSA)**.
- *Guidelines* that we have learned when applying the principle of MDD to our case study. The objective here being to highlight important **differences between theory and practice** as a guide to developers interested in adopting MDD.
- In particular, we offer guidelines for overcoming the observed limitations of RSA with respect to model-code synchronization.

In addition to being a comprehensive assessment of MDD through RSA, as a secondary contribution we also

- Investigate the combination of Test Driven Development (TDD) [Astels 2003] with MDD.

1.4 Outline

This thesis is organized into six chapters and one appendix. Chapter 1 introduces Model Driven Development, the main problems that have to be resolved and the major contributions of this thesis. Chapter 2 provides brief background information of our work. We introduce the modeling language UML, the MDD and TDD methodologies followed by the topic of refactoring. We then provide a summary of the related work with respect to this thesis. Chapter 3 provides the scope within which RSA is used and a brief description of the key features of RSA. It also explains the technical details of applying MDD with RSA. Chapter 4 describes the main contribution of this thesis namely, a detailed case study in MDD. The subsections describe the iterative process applied to the case study. Chapter 5 describes the major limitations of RSA with respect to MDD, especially the problem of model-code synchronization. It also offers guidelines based on the lessons learnt while applying MDD to the case study. Chapter 6 provides the conclusion of our thesis and discusses future investigations that can be carried out to build on our work. The Appendix lists various technical details regarding RSA features and customizations made to the IDE to implement the guidelines as suggested in Chapter 5.

2 Background and Related Work

In this section, we present background material that is necessary for the understanding of the core contributions of the thesis. We first introduce the Unified Modeling Language (UML), which is used extensively for modeling in this thesis. This is followed by introductions to Model Driven Development (MDD) and Test Driven Development (TDD), the two approaches followed in this thesis. We then cover the topic of refactoring, followed by related work.

2.1 Unified Modeling Language (UML)

2.1.1 UML and Modeling

The Unified Modeling Language (UML) defines a suite of visual notations suitable for expressing business processes, system and software requirements as well as design, implementation and deployment scenarios. UML is particularly suited for the description of software systems developed in the Object Oriented (OO) style. There are different ways to approach modeling with UML which have been classified into two main categories (see Figure 1):

- **Dynamic Modeling.** Dynamic modeling is performed to analyze business processes, understand user behaviour, or design the interactions between software entities. In UML 2, dynamic modeling is performed using Behaviour Diagrams.
- **Static Modeling.** Static modeling is performed to organize the structure of the various software components. In UML 2, static modeling is performed using Structure Diagrams.

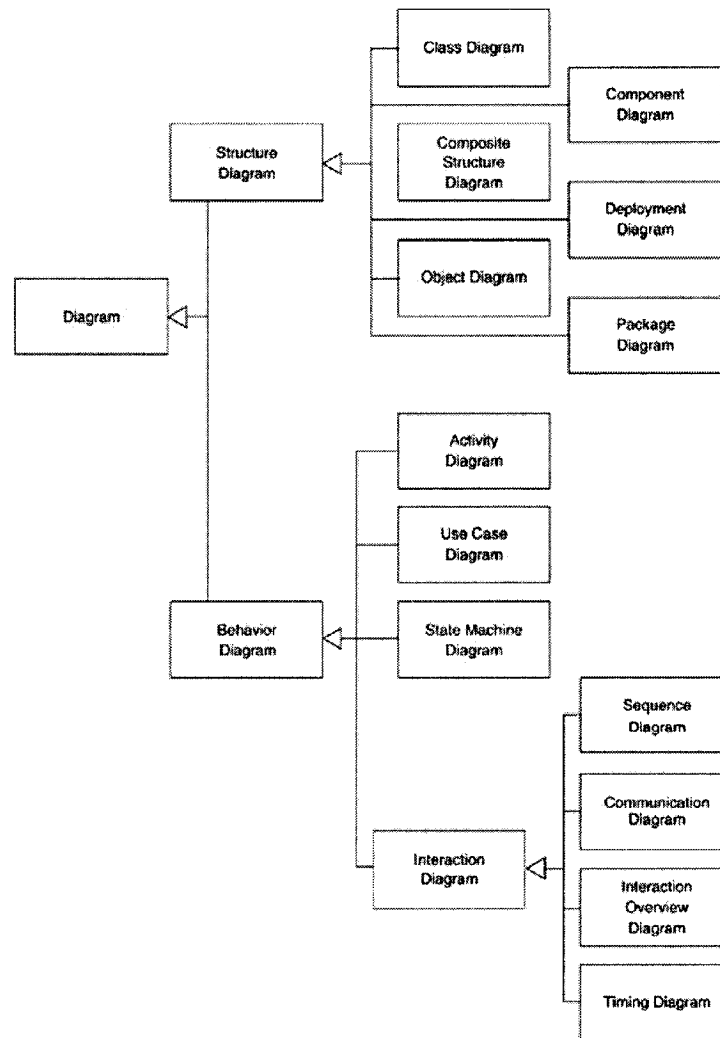


Figure 1: Types of UML Diagrams [Fowler 2003]

Though there are many types of diagrams in UML, we shift our focus to the UML Class diagram, as it is the diagram type used most frequently in the thesis. A class diagram is used to describe the static structure of classes and interfaces, and the static relationships between them. The static structures of, and relationships between classes are represented through UML Properties and UML operations. In a UML class diagram, a class is shown as a rectangular box usually with three horizontal compartments (see Figure 2). The horizontal compartments from top to bottom give the class name, class properties and class operations.

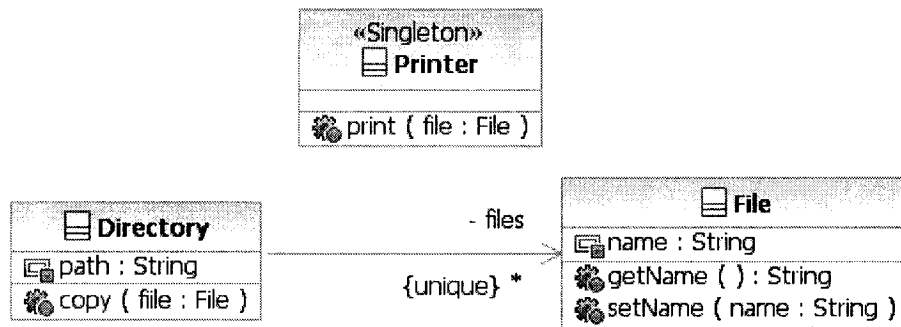


Figure 2: Class Diagram explaining basic UML concepts

2.1.2 Properties

UML properties denote the structural features of a class i.e., its fields. UML Properties can be shown in two different ways in a UML class diagram: UML Attributes, or UML Associations. An attribute is shown within the class box in its second compartment. The syntax for an attribute is of the form

```
visibility name: type multiplicity = default-value {property-string}
```

- **visibility:** describes the access rights to the property. The symbols +, -, ~, or # representing public, private, package or protected access, respectively
- **name:** the attribute name. This is the only mandatory field.
- **type:** class or primitive type of the attribute.
- **multiplicity:** the number of instances contained in this property.
- **default-value:** the value assigned to the attribute the moment an object is instantiated.
- **property-string:** additional qualification(s) given to the attribute

For example

```
- title:String {1} = "NO_TITLE" {readOnly}
```

defines a private attribute 'title' of type `String` that can contain only one instance of type `String`. It has a default value of "NO_TITLE". The `readOnly` property signifies that its value cannot be modified, but only read.

In the case of properties, a UML association is normally a solid, directed line from the class that owns the property to the class that indicates the type of the property. Figure 2 shows that class `Directory` has an

association with name `files` of type `File` and a multiplicity of more than one. The name of the property as well as its multiplicity is mentioned near the directed end of the association. Normally, attributes are used to represent common data types like `String`, `Boolean` and `Date`, where the value of the object is more important than its identity. Associations are used to represent a complex type like `Person`, `Product`, etc., where the identity is more important than its value [Fowler 2003].

2.1.3 Operations

A UML operation is an action that a class or interface can perform. It is distinct from the term *methods* in that an operation is a declaration (or method signature) of what the class can do and a method is the implementation of how a class chooses to perform such an action. This distinction between an operation and a method is associated with polymorphism whereby if three classes implement an interface, then one interface operation may have up to three different methods corresponding to one for each class. The syntax for a UML operation is of the form

```
visibility name (parameter-list): return-type {property-string}
```

- **visibility:** : describes the access rights to the operation. The symbols `+`, `-`, `~`, or `#` representing public, private, package or protected access, respectively.
- **name:** method name. This is the only mandatory field.
- **parameter-list:** list of parameters for the operation. The syntax is similar to those of properties.
- **return-type:** the type of the value returned by the method.
- **property-string:** method qualifiers.

For example

```
+ getName():String {query}
```

declares an operation named *getName* with public access that returns an object of type `String`. The method is qualified as a *query* method [Meyer 2000]; i.e., it does not modify the state of the object on which it is invoked.

2.1.4 Constraints

UML constraints define restrictions on UML elements. An example of a restriction might be that a UML property of multiplicity greater than one should only have unique instances. All UML constraints are placed between curly braces. Figure 2 shows that the association `files`, of class `Directory`, should refer to only unique instances of class `File`. These constraints can be expressed in a natural language, a programming language, or UML's formal Object Constraint Language (OCL).

2.1.5 Stereotypes

The basic building blocks of UML (i.e., classes, properties, operations, and constraints) can be extended to capture domain concepts or common design solutions. Extensions allow the domain concepts or design solutions to be also used as building blocks, thereby supporting better abstraction of the problem domain and enhancing design reuse. These extensions are called UML profiles. One of the extension mechanisms for defining UML profiles is the stereotype. Stereotypes introduce domain concepts and design solutions into the vocabulary that will be used for modeling. Figure 2 shows that class `Printer` has been stereotyped as a Singleton [Gamma et al. 1995] as indicated by the word appearing between *guillemets* (i.e., << and >>). Stereotyping relieves us of having to specify methods like `getInstance()` which are understood to be implied by the stereotype.

2.2 Model Driven Development (MDD)

2.2.1 The Need for Models

Before introducing Model Driven Development (MDD), it is necessary to list the motivations for considering this paradigm in software development. Most of the motivations stem from the limitations of current third generation languages (3GLs) for developing complex, distributed systems. The main limitations that are relevant within the scope of this thesis are:

- 3GLs are normally used to express the implementation details of software development, i.e., the technical domain. However, the major complexity in developing large-scale complex systems lies in understanding the problem domain for which the software is being created [Evans 2003]. This understanding serves as a basis for software design, independent of the user interface or any other

technical infrastructure (which may be implemented in 3GLs). Thus, the technical nature of 3GLs makes them inadequate for bridging the wide semantic gap between the technical domain and the problem domain [France & Rumpe 2007].

- Even in the technical domain, the detailed syntax of 3GLs makes them too verbose for effectively communicating the high-level design decisions of large-scale systems.

These needs for better expression of the problem domain and high-level design have led to the usage of graphical modeling languages like the Unified Modeling Language [UML] where developers reason and communicate about the system to be built using model abstractions. In such models, the visual notations are expressive enough to describe the underlying concepts of the problem domain, and yet hide the implementation details of the design solution. At the same time, these models are formal enough to be similar to the concepts in a technical domain so that they can be mapped to an implementation structure without much effort.

2.2.2 Essentials Components of MDD

The above-mentioned value of modeling has given rise to MDD where models are considered as the primary artifact of software development. There are two essential components in MDD: the meta-model and model transformations.

In MDD, a model's syntax and semantics is made precise by defining its meta-model (i.e., a model of the model). This is important because it is necessary to ensure that the same graphical symbols are not interpreted differently by users. Having a meta-model also facilitates communication between stakeholders. For example, if a stakeholder uses the term *package*, other stakeholders can refer to the meta-model and understand that a *package* refers to a model element that groups model elements of all types including other packages. In addition, perhaps the most practical use of a meta-model is that tools can generate code from models using the meta-model.

In fact, the ability of tools to transform models to code via automation is a critical factor for the success of MDD. A transformation is formally defined as the process of taking one or more source models as input and generating one or more target models as output with the help of transformation rules [Sendall & Kozaczynski 2003]. It is the transformation rules that enable possible automation of activities like reverse

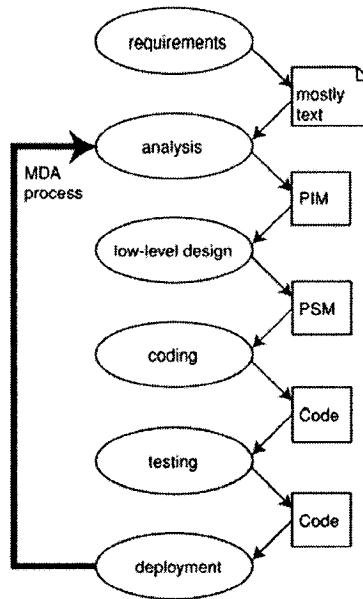


Figure 3: MDA software development life cycle [Kleppe et al. 2003]

engineering for code synchronization, application of patterns and refactoring. Though transformations can take care of almost all code generation activities, it is important to allow extensibility of transformations. This is because different teams have different needs from the same transformation and prefer tuning it to their own requirements. If transformations cannot be extended, then developers prefer not using transformations at all [Weis et al. 2003].

2.2.3 Model Driven Architecture (MDA)

One of the popular instances of MDD is called Model Driven Architecture (MDA) and is defined by the Object Management Group [OMG]. MDA is an MDD framework where the modeling language is the Unified Modeling Language [UML]. Figure 3 shows the software development life cycle as prescribed by MDA.

In the MDA software development cycle, the analysis phase produces a Platform Independent Model (PIM). The PIM is used to model concepts as close as possible to the problem domain, abstracting away from the technical platform details of the implementation. For example, the problem domain of healthcare industry can be modeled using the PIM. Whether the software project is developed, using Java or C#.NET does not matter at the PIM level. Once the PIM is created, we can use transformations to obtain a technical platform specific abstraction, i.e. the Platform Specific Model (PSM). The technical platform could be Java

EE or .NET. It could also be a client server system or a mainframe. The PSM is used to model concepts close to the technical domain but at a level where it is still able to capture the high-level design aspects of the technical domain. The PSM is then transformed to partial or complete code.

2.2.4 MDD Vision

If the vision of MDD is realized, the following gains can potentially be made in the field of software development:

- **Productivity:** Models can be used to generate partial or complete software systems using automated transformations provided by tools. Best practices in coding can be captured in these automated transformations, thus potentially improving developer productivity and software quality. In fact, current code generated by tools is comparable to hand written code in terms of efficiency [Selic 2003].
- **Maintenance and Documentation:** Since software development is driven by models, the high-level design documentation is always synchronized with code. This would be very useful for long-lived projects where the developers who initially developed the software are replaced by other developers (“maintenance developers”) who will have up to date documentation of high-level design.
- **Knowledge Reuse:** Considering the importance of capturing domain and design knowledge, it would be very productive if the same knowledge can be captured once in an implementation independent model and tools use the model to generate code in various technological platforms like Java EE or .NET.
- **Validation:** If the model is precise enough, it can be analyzed mechanically for desirable or undesirable characteristics. By detecting and resolving the undesirable characteristics in the model itself, we can analyze potential design decisions before investing time and resources on implementation and testing.

2.3 Test Driven Development (TDD)

Test Driven Development (TDD) is a software methodology that advocates writing tests first and then writing code to satisfy those tests. These tests are called Programmer Tests [Astels 2003]. Programmer

Tests are unit tests with the difference that their primary aim is to determine what code is to be written, in addition to testing the code for different scenarios.

2.3.1 Testing Frameworks

Unit/Programmer Tests are normally written using testing frameworks like JUnit [Hunt & Thomas 2003] that help the programmer in creating automated tests. These frameworks enable maintaining a large suite of tests and running all of them together with less effort. Programmers can test their implementations by using assertion methods provided by the framework. These assertion methods generally compare the actual output from the implementation and the output expected by the test. If the expected and actual output do not match, then the assert method throws an error which is caught by the framework and reported to the programmer. Thus, with the help of assert methods, a programmer can validate if the implementation provides the expected behavior. The framework also provides mechanisms to reuse common code (also called fixtures) that should be run before and after every test. In JUnit, the common initializing code is written in a method called `setUp()` and the code required to dispose of the fixture is written in a method called `tearDown()`.

2.3.2 Methodology

When a feature is to be added to the software, the programmer thinks of the tests required to validate the presence of the feature in the implementation. The programmer writes a test and executes the test. Since the feature has not been implemented yet, the test fails. The programmer then writes the minimum code to satisfy that test. If the test succeeds, the programmer writes another test and enhances the code just enough to satisfy this test and no more. In this manner, the programmer works in a cycle of test and code, incrementally developing the feature for the software. As tests are written incrementally, soon there will be a large collection of automated tests. These tests may have common programming problems like code duplication, long methods, etc. and hence have to be refactored. For example, duplicate code can be refactored to fixtures supported by the testing framework.

2.3.3 Advantages and Disadvantages

- By writing tests first, we ensure that for every feature that goes into production code there are tests to verify its behavior.
- Tests and implementation are developed concurrently when the context of the feature is fresh in the programmer's mind. Thus, the collection of tests has a greater likelihood of being comprehensive and relevant to the feature.
- Automated testing encourages repeated validation of code changes. When used consistently after every code change, the programmer can instantly detect the offending change. As opposed to this, the programmer may make a large set of changes collectively. However, if a bug is detected, the programmer may spend additional time trying to figure out which change introduced the bug.
- Refactoring code is essential for successful maintenance of systems. The safety net of tests allows one to refactor code without the fear of breaking a feature or some other dependent feature. An incorrect refactoring can be immediately caught by running tests and the feature can be reverted to the version before the refactoring.
- As tests drive implementation, one only designs what is necessary to satisfy those tests. This prevents the introduction of additional code that may not be used later. Such a restriction on design keeps code simple and relevant.
- The TDD methodology requires a different mindset as compared to traditional methodologies, and so there is a slight learning curve associated with TDD.
- Though TDD keeps code simple and relevant, some industry practitioners find that a lack of upfront design is a hindrance to its adoption [George & Williams 2003].

2.4 Refactoring

[Fowler 2000] defines refactoring as “a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.” Managers would typically resist such changes, as they do not want to change code that works and the time spent changing code could be spent writing new features, thus gaining more revenue for the company. However, refactoring cannot be neglected because of the following factors:

- Every time we add new features to the software without considering its impact on the overall design structure, we may possibly introduce unfavorable aspects of programming like duplicate code or long methods/classes that perform multiple responsibilities. Ad hoc addition of features will result in the decaying of the system design to such an extent that it will become infeasible to add any more features without breaking the system. Refactoring the design at regular intervals cleans it up and makes it more adaptable to change. For example, by eliminating duplicated code, we ensure that if changes are to take place, only a single section of code needs to be touched, instead of having to modify multiple places, which is error prone.
- The reality of software development is that code may be developed by one developer but it has to be maintained by many developers. Many developers, after a short while, cannot understand even their own code, and rediscovering the design of others' wastes precious time. One of the driving purposes of refactoring is to write code that is easily understood by other developers. Code that is easy to understand better reveals the intent of the design. The code also becomes easier to modify when requirements change or bugs are reported.

Nevertheless, managers still may not be convinced. Since refactoring involves making changes to software, it can also break a previously working system. This concern can be relieved by maintaining a suite of automated tests, which can be performed quickly and effortlessly after every change. These tests will quickly point out if the change is faulty, and we can switch back to the previous version of working code. Refactoring without tests can lead to additional bugs in the system or cause a failure in another part of the system.

Refactoring can be further divided into two different approaches depending on the level of abstraction:

- Refactoring to Code [Fowler 2000]: These are low-level refactorings made to code. They mostly involve minor changes like renaming variables and methods to be more meaningful or moving methods from one class to another for more cohesion and less coupling.
- Refactoring to Patterns: These refactorings are performed to improve the design of the software. The source design model is transformed into a target design model resembling a design pattern [Gamma et al. 1995], which embodies best design practices. Sequences of low-level refactorings are performed to

apply the design pattern in our design model. After every low-level refactoring, we still run tests to verify that the changes made work. These sequences of refactorings are also called “composite refactorings” [Kerievsky 2005].

In order to start refactoring, we need to know what code to refactor. Catalogs of refactorings have been published to improved code and design [Fowler 2000, Kerievsky 2005]. Most common problems arise due to duplicated, unclear, and complex code. These problems are also called “bad code smells” [Fowler 2000] which signifies that certain pieces of code do not feel right to the intuitive nature of good developers.

A simple adapted example from [Nilsson 2006] can explain how to find a code smell and refactor code. This code is an example of software built for a mining company. The purpose of the code is to determine the number of trucks required to carry extracted asphalt from the mills. The code piece in Java is

```
public int numberOfTrucksRequired(){
    return millingCapacity/(truckCapacity*60/unloadingTime);
}
```

Figure 4: Example of Code that has a “bad smell”

The formula in the code, though simple, does not reveal well its intent. It takes some time to understand the reasoning behind the formula. Using a refactoring called “Extract method” [Fowler 2000], we extract the formula into a method which expresses the intent better. The refactored code is shown below.

```
public int numberOfTrucksRequired(){
    return millingCapacity/singleTruckCapacityDuringOneHour();
}
```

Figure 5: Example of Code after Refactoring

As it can be seen, understanding the code becomes much easier, and we do not waste time trying to figure it out the next time around.

While refactoring, several best practices can be followed. They are

- Undertake refactoring in small safe steps. After every small refactoring, run tests to verify that the change is valid. Making small incremental changes is faster than performing a set of refactorings together. When we make many refactorings together, we could introduce errors. We would spend a considerable amount of time determining the location of the mistake and trying to fix it.

- Perform code reviews. Code reviews allow multiple developers to inspect the code, thus providing insights that the author of the code may have missed. Thus, extreme programming [Beck 1999] encourages pair programming where two developers work together.
- Make code readable. At many places, complex code can be replaced by code that is nearer to a natural language. This abstraction makes it easier to understand code and change it, if required.
- Make refactoring a continuous habit. By refactoring at regular intervals, the code is kept clean thus making addition of features and solving bugs that much easier. The more we procrastinate about refactoring, the messier the code becomes and the more difficult making changes to code becomes.

Even though refactoring aids in understanding code and managing complexity, critics can argue that refactoring often increase the number of classes and methods which may decrease the performance of the software. This could indicate that if performance is a prime concern, then refactoring should not be done. Apart from real-time systems, there are applications that may not require such stringent performance requirements. For such applications, it is appropriate initially to design for better understanding and ease of change. Refactoring should be done as required without considering performance.

Once the design is stable and there is a working application, the developer can consider the performance of the system. Most studies of software performance [McConnell 1999] state that the majority of performance slow downs are caused by a minor section of code. It has been found that 20% of a program (also called hot spots) consumes 80% of the execution time [Boehm 1987]. By tuning these hot spots, we can obtain a huge improvement in performance. These hot spots can be detected by using a performance profiler. The profiler will detect the code sections that are frequently invoked, take a lot of time to execute or use a lot of memory. These code sections can then be optimized for better performance, giving the developer more value for time spent. Additionally, virtual machines for platforms like Java detect hot spots during program execution and generate highly optimized native code for them. If the virtual machine detects that a method is invoked repeatedly, it will inline the native code for that method in the native code of the invoking method. This reduces the overhead of method invocations and provides the virtual machine with a larger section of code to optimize [Java HotSpot]. This further reduces the performance impact of refactoring.

Thus, apart from virtual machine optimizations, if the developer has to focus on finding the code sections that have to be tuned, it will be very difficult if the design is not understandable and flexible. Refactoring at regular intervals allows developers to create a design that makes it easier to pinpoint the region of change and make modifications. In this manner, we can have the best of both worlds, well-designed software with minor tweaks in code for better performance.

2.5 Related Work

MDD has been used in a case study for the development of a software platform for a new generation of digital TV set top boxes [Mattsson et al. 2007]. The project requirements were to develop a software platform with a short time to market for a continuously evolving hardware platform. It was understood that the platform would have to be maintained and adapted to various markets, with stringent quality and performance criteria. The case study found MDD to be inadequate for capturing all the architectural constraints that go beyond structure. Though the architects had created an executable architectural framework, they still needed to use natural language to express rules for the usage of the architectural framework in order to design components in the architecture. The rules were hard to understand and were often interpreted differently by developers. The rules could not be enforced, and hence the architects had to do manual reviews, which took a significant part of their time. Since the architects were busy, there was a scarcity of resources to review the code with the result that the development was error prone. In addition, when the architecture changed, there was major rework regarding the architectural constraints added manually. Thus, the authors advocated support for the modeling of architectural rules, coupled with automatic enforcement of the architectural rules in the generated models.

Case studies highlighting the adoption of MDA in the industry have also been published [Guttman & Parodi 2007]. These case studies describe all aspects of the project using a business focus.

3 Rational Software Architect (RSA)

In this chapter, we highlight the key features of IBM Rational Software Architect (RSA) for the application of MDD. RSA serves as a tool not only for modeling but for implementation as well. RSA is built over the extensible Eclipse Platform [Eclipse] that provides the basic IDE framework for software development.

3.1 Scope of Usage of RSA

For this thesis, we investigate the use of RSA within the following scope.

- This thesis explores the application of MDD with RSA by a single developer only. The use of RSA by teams is outside the scope of this thesis.
- RSA is only used for creating design models. It is not used for creating use cases or other analysis artifacts.
- Though MDD can involve the creation of both static and dynamic model artifacts, the use of RSA is limited only to static UML class diagrams. In other words, this thesis is restricted to the study of maintaining model-code synchronization and communicating design intent using class diagrams in RSA.
- Complete code generation is outside the scope of this thesis. For this case study, the basic transformation functionality that generates methods signatures only is used. In such transformations, the implementation details have to be fleshed out in the method body by the developer.
- The usage of RSA for the thesis is also confined only to the Java Platform.

3.2 Key RSA Features for MDD

Before we explain the key features of RSA for MDD, it would be useful to compare the MDD methodology used by RSA with the MDA framework. As explained in section 2.2.3, MDA supports the creation of a Platform Independent Model (PIM), which is translated to a Platform Specific Model (PSM). The PSM is then translated directly to code. The modeling language used in MDA is UML.

Though RSA employs UML as its modeling language for MDD, it does not strictly adhere to the MDA framework. Instead of creating an explicit PSM from PIM, RSA allows the user to transform PIM directly

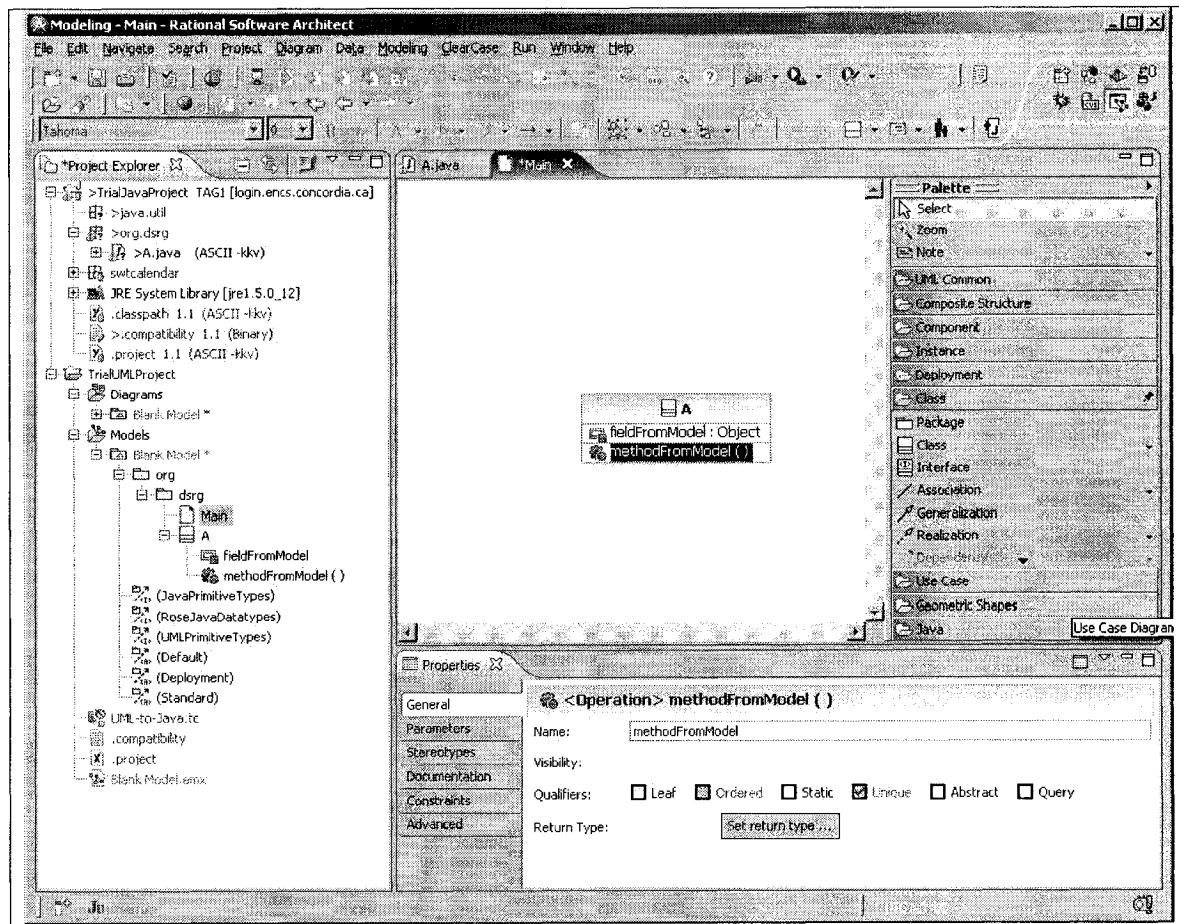


Figure 6: Modeling Perspective in RSA

to code. To compensate for the lack of a PSM, RSA provides the feature of code visualization as UML diagrams which provides features like refactoring [Swithinbank et al. 2005]. Therefore, in RSA, the concept of PSM is absent.

3.2.1.1 Modeling Support

The creation and management of UML models and all related artifacts is performed in the Modeling Perspective in RSA (Figure 6). A Perspective as defined in Eclipse is a customized layout of windows, menus, and actions in the IDE for the efficient execution of a particular software activity. For example, the Debug Perspective contains all the windows, menus, and actions that are relevant to the debugging of source code.

The Modeling Perspective contains a set of views that makes working with models easier. The Project Explorer (left of Figure 6) is the main view used to navigate within models. The user can select model

element attributes and related UML diagrams as well. UML diagrams are created and edited in the Diagram Editor (centre of Figure 6 titled as *Main*). To the right of the diagram editor, RSA provides a quick reference to the list of UML model elements in a palette (right part of Figure 6). Model elements like classes and packages can be dragged and dropped from the palette to the diagram editor. If a model element is selected in the Project Explorer or diagram editor, all its information is displayed in the Properties view (lower section of Figure 6). RSA also supports the modeling of Java primitive types and Java library classes directly into the UML model. This is possible by importing the required model libraries into the particular UML model (see the Appendix).

3.2.1.2 Transformations

As explained in section 2.2.2, a transformation maps a source model into a target model using transformation rules. A transformation instance for a particular source and target model is called a transformation configuration. Along with the source and target model, the user can specify other transformation properties specific to these models. For example, by selecting an option in the transformation configuration, the user can choose to auto generate getters and setters for class attributes.

With respect to Java, RSA 7 provides transformation from UML to Java 1.4 and UML to Java 5.0. The source model for such transformations is the UML model. The target model is the implementation. In this thesis, a transformation from UML to a target model is called a forward transformation.

When a class is created for the first time by forward transformation, the members of the class are appended with an `@generated` tag (see Figure 7) within the Javadoc comments before every member (see Figure 8).

```
@generated "UML to Java V5.0  
(com.ibm.xtools.transform.uml2.java5.internal.UML2JavaTransform)".
```

Figure 7: The mechanism used by RSA to maintain a link between model and code

This tag is significant because if the forward transformation is rerun, then all qualifiers and members of the class along with qualifiers, parameters, and methods bodies are overwritten. If the user wishes to prevent the overwriting of a class or its class members, then the particular `@generated` tag should be removed. However if the tag is removed, then the link between that class member or class and the corresponding element in the UML model is lost (i.e., if an element in the UML model is deleted and a forward transformation is rerun, the corresponding element in code is not deleted if the `@generated` tag has been

```

/**
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @author r_abrah
 *
 * @generated "UML to Java V5.0 (com.ibm.xtools.transform.uml2.java5.internal.UML2JavaTransform)"
 */
public class A {
    /**
     * <!-- begin-user-doc -->
     * <!-- end-user-doc -->
     *
     * @generated "UML to Java V5.0 (com.ibm.xtools.transform.uml2.java5.internal.UML2JavaTransform)"
     */
    private Object fieldFromModel;

    /**
     * <!-- begin-user-doc -->
     * <!-- end-user-doc -->
     *
     * @generated "UML to Java V5.0 (com.ibm.xtools.transform.uml2.java5.internal.UML2JavaTransform)"
     */
    public void methodFromModel() {
        // begin-user-code
        System.out.println("Some code written manually after transformation" +
            "And placed within the inline comments");
        // end-user-code
    }
}

```

Figure 8: Snapshot of generated code with @generated tag and inline comments

removed). However, after creating a class for a first time after forward transformation, the user may just write the method bodies without changing the class or method structure. In this case, the user can prevent the overwriting of method bodies by placing the method body between the comments `// begin-user-code` and `// end-user-code` (see Figure 8). Instead of the user manually entering these comments, they can be auto generated by customizing Eclipse Preferences as explained in the Appendix.

RSA also supports transformations where a Java Project is the source model and a UML model is the target model. In the thesis, this transformation is called reverse transformation. During reverse transformation, a temporary UML model is created from the source model (i.e., the Java Project). This temporary UML model is then automatically compared with the target UML model by RSA. The user inspects both the models and accepts or declines changes from the temporary model.

An example of such a model merging is shown in Figure 9. The temporary model is shown in the left half with differences in bold. The target model is shown in the right half. In this example, a method `methodFromCode()` has been added in the Java Project before reverse transformation. RSA already selects the checkbox associated with the method to indicate that it is going to be added to the target UML

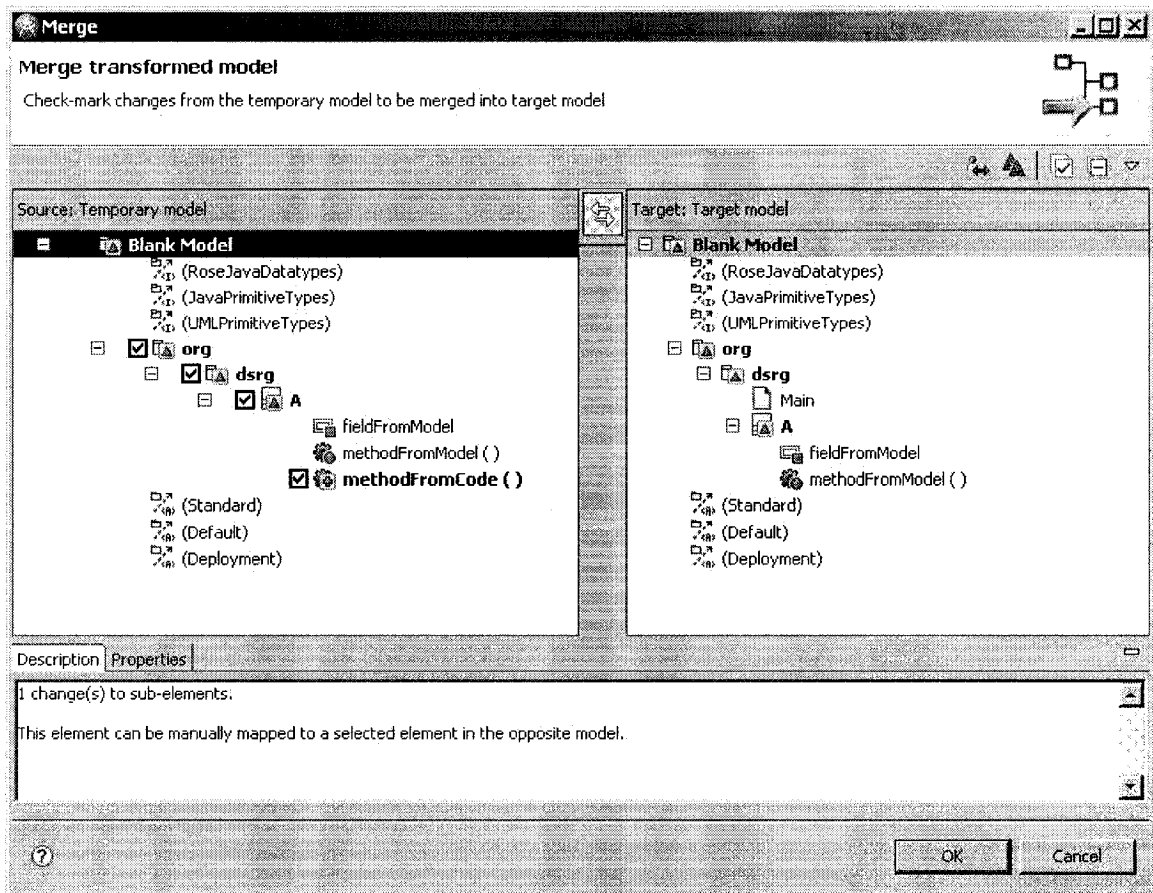


Figure 9: Model Merging Window during Reverse Transformation

model. The user confirms whether it should be added or not and then the changes are merged into the target UML model.

In addition to modeling and transformations, RSA also provides a feature to validate models. In order to use the model validation for transformations to Java, RSA provides transformation profiles that can be applied to the model. These profiles help validating the models before transformation or provide warnings after the transformation is executed.

4 Case Study: Development of a Project Management System (PMS)

In this chapter, we describe the application of MDD and TDD to the development of a Project Management System (PMS), where the term “project” is taken to encompass many different sorts of undertakings (not only software development projects). This chapter is organized as a series of iterations, each describing the evolution of the PMS with respect to MDD.

4.1 Iteration Presentation Style

In the iterations where we apply MDD, we perform frequent cycles of modeling, forward transforming the models to code and filling in the implementation details. If any structural changes (e.g., adding or removing class members) are made in the code directly, then reverse transformation from code to model is performed. In this manner, the model represents the up-to-date static information of the code (i.e., the model and code are always synchronized).

However, the thesis describes all MDD iterations with a discussion of the design rationale for each feature followed by the corresponding UML model class diagram. The associated implementation and tests are not described in this thesis. This description style is chosen to demonstrate how developers can reason about software development at a higher level with the help of models and abstract away from the implementation details. The description of the design rationale also demonstrates that the case study is not contrived. However, for a practical demonstration of MDD, a sample of the cycle of modeling, forward transformations, and implementation is presented in section 4.3.

4.2 Iteration 0: Preliminary Analysis and Creation of a Domain Model

In this iteration, we define an initial domain model that drives development. Initial domain analysis with users gave rise to the following domain concepts:

Task: The main goal of the system, according to the users, is to capture project related actions. Since the term “action” is too generic, we asked the users to clarify the nature of their actions. Common actions for users include attending meetings, reading, and writing papers, coordinating courses and graduate student

projects. Possible alternatives like “Activity” and “Task” arose during the discussion. We settled on “Task,” as it is a term commonly associated with project management and it is familiar to users.

Status: Each task has a completion status. Initially, the possible values defined are NOT STARTED, IN PROGRESS and DONE.

Importance Level: Tasks have different levels of importance. Importance levels are defined as A, B and C, with A indicating the most important.

Description: Information that explains the task in more detail or supports its execution.

Folder: Users normally have huge collections of tasks. They wanted a mechanism to manage their tasks more effectively. To reduce information overload, a user suggested grouping related tasks by categories. Categories could also be divided into subcategories, allowing further division of tasks. However, users also have a requirement to store tasks and related documents or other artifacts together. In this regard, a category is more a concept used to group entities, not to store artifacts. The storage of artifacts is normally associated with folders (a metaphor commonly used in software applications including operating systems). The hierarchical decomposition provided by categories is also present in folders, as folders can have subfolders. Therefore, the folder metaphor was chosen as a means of storing tasks and related artifacts.

A very brief list of the features requested by users includes:

- Create Task
- Edit Task
- Group Tasks using Folders
- Add / Remove Tasks from Folders
- Create / Remove Folders
- Create a Hierarchy of Folders
- View Tasks in a Folder and Subfolders
- Sort Tasks by Task Attributes
- Reorder Tasks in User-Specified Order
- Share Tasks between Folders

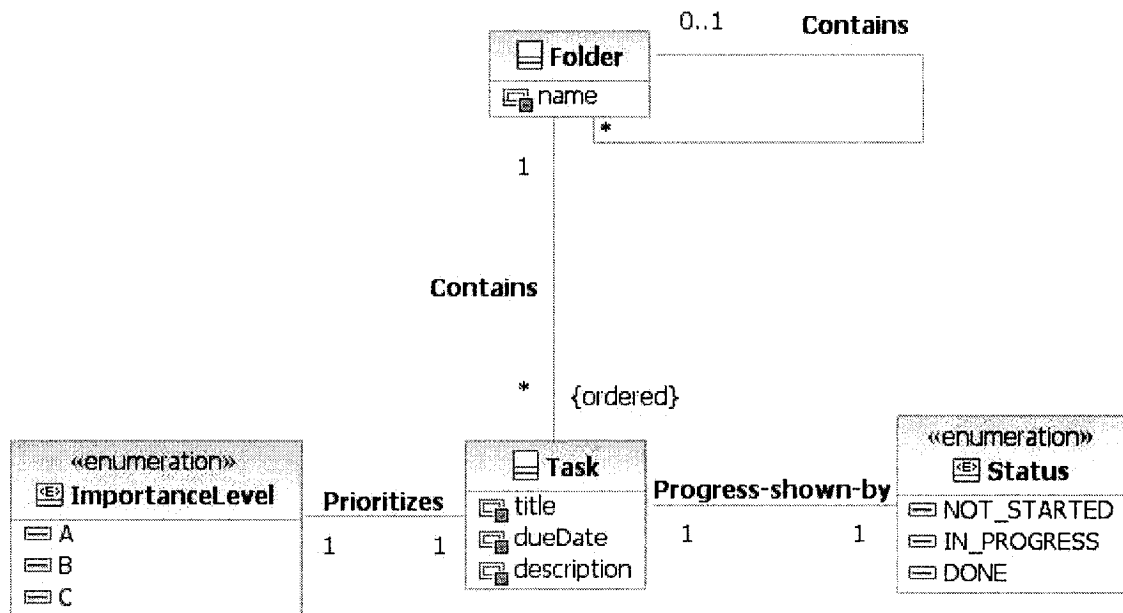


Figure 10: Initial Domain Model

For initial iterations, a task can be contained in a single parent folder only. This helps simplify the domain logic and hence allows us to complete initial iterations more quickly so that user feedback can be obtained. Since users would like to order tasks, we explicitly qualify the tasks to be ordered in the domain model. This is because collections in UML, by default, are assumed to be unordered and have unique elements [Fowler 2003, p. 38].

Most folders have a containing folder, and those that do not are called root folders. Note that we restrict each folder to have at most one containing folder. This is because if a folder were shared by multiple folders, it could lead to folder loops, which would need additional logic. The restriction on sharing may be reassessed later when the domain concepts have been realized in an initial implementation. All the concepts explained above are presented in the domain model shown in Figure 10.

4.3 Iteration 1: MDD for Basic Features

In the previous iteration, we created an initial domain model that defines the main concepts of project management. In this iteration, we transform the initial domain model into a design using MDD. This design is then implemented and tested. The following are the features that are created in this iteration:

- Create/add task
- Delete task
- Edit task
- Create/add folder
- Delete folder
- Edit folder

The rationale behind choosing these features for implementation is that they form a very basic set of features and they are very similar in functionality. Hence, the effort taken to create one feature would carry over to similar features. For example, the logic required to add a task and to add a folder would be very similar. We manually created an initial design class diagram (see Figure 11) with the help of the domain model. To keep the diagram simple, we do not show the accessors and mutators of class properties.

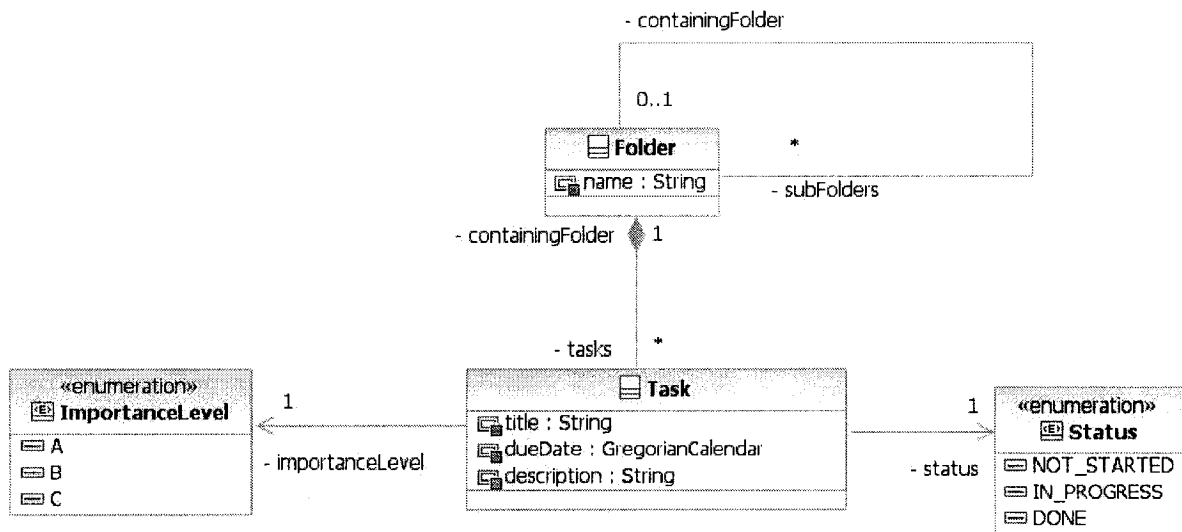


Figure 11: Initial class diagram

4.3.1 Domain Model: Order of Tasks and Folders

As indicated in the domain model, users require tasks within a folder to be ordered. This characteristic of the domain model is considered before the basic features mentioned previously because it influences the decision concerning the type of collection that should be used to hold tasks.

In general, UML class properties¹ can be decorated with qualifiers as is shown in Figure 12. In particular, we can constrain collection properties (i.e., properties declared with a multiplicity greater than 1) as having *unique* and *ordered* elements. Such qualifiers have an impact on the transformation process from UML models to code. Details about the transformation, along with the source UML model and the target Java project, are specified in a transformation configuration file. One of the details that can be specified in a transformation configuration is the type of Java collection to be associated with collection properties. An example of a transformation configuration for the `Folder.tasks` property is shown in Figure 13. In particular, what is shown is the *Collections* tab, which defines the mapping between the UML collection properties and the corresponding Java collection types.

¹ Recall that a UML class property is expressed as either a class attribute or a class association.

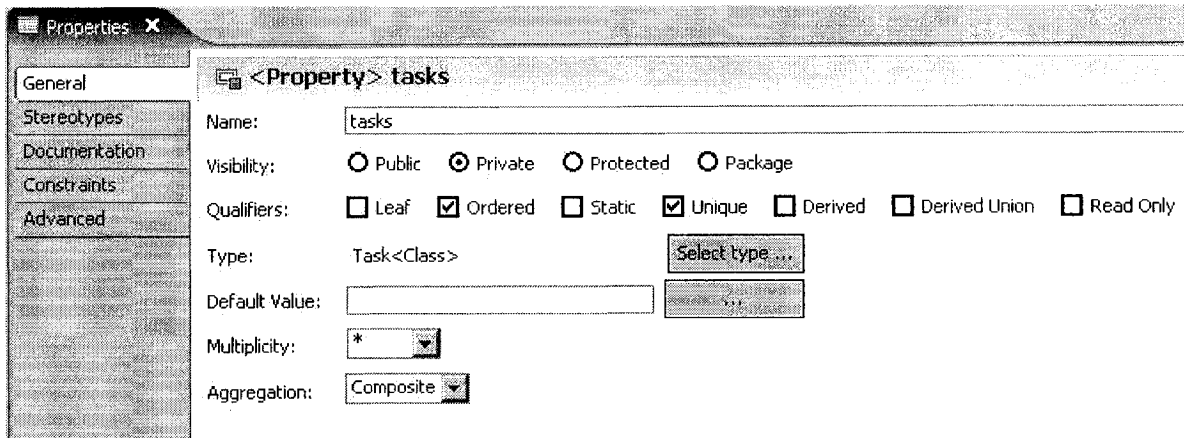


Figure 12: Snapshot of properties for `Folder.tasks`

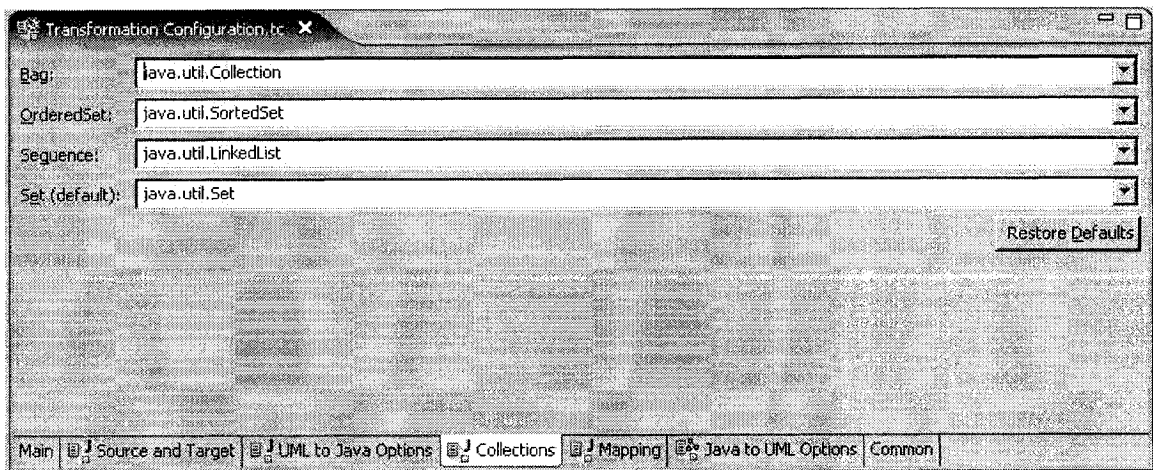


Figure 13: Collection tab in transformation configuration file

As shown in Figure 12 and Figure 13, if we require the collection `tasks` to have ordered and unique elements, RSA provides a `java.util.SortedSet` as a possible collection type. Elements in a `SortedSet` can be ordered using an implementation of `java.util.Comparator<T>` (Java 5 API). The tasks could be compared based on a `rank` property in `Task`. We could alternatively use a `java.util.List` or an array. A `List` provides ordering, but fails to enforce uniqueness of instances. We could enforce uniqueness by controlling the addition of tasks to the list.

Using a `java.util.List` seems to be the simpler solution; however, one of the features requested by the user is to see all tasks in a folder (including the tasks in its subfolders) and be able to reorder all these tasks relative to each other. This means that the user could choose to reorder a subfolder task before all of the folder tasks. Since the folder only knows about tasks residing within it, it cannot determine if any subfolder

tasks precede its own tasks. Since using a simple list is inadequate, we chose to utilize the `java.util.SortedSet` functionality by creating a rank attribute in class `Task`. We create a private field `NEXT_RANK` in class `Task` to assign a rank when a task is created (see Figure 16).

Once a rank is assigned, we make use of a `RankComparator`, which implements the `java.util.Comparator<T>` interface for the ordering of tasks according to their rank. During the creation of every sorted set tasks in class `Folder`, we have to assign a `RankComparator`. Creating multiple instances of `RankComparator` is redundant, as it does not possess state information. Therefore, we make it a Singleton [Gamma et al. 1995].

RSA supports modeling and partial code generation of the design patterns. Figure 14 shows the “Pattern Explorer” view on the left that presents all the design patterns. We select the Singleton Pattern from the pattern explorer, drag and drop it to the class diagram “Domain Model Pattern” on the right to obtain a “Pattern Instance” Singleton. We drag and drop the class `RankComparator` into the Singleton pattern instance. This stereotypes `RankComparator` as a Singleton as shown in Figure 14. RSA also generates code for the Singleton pattern during forward transformation from model to code (see Figure 15).

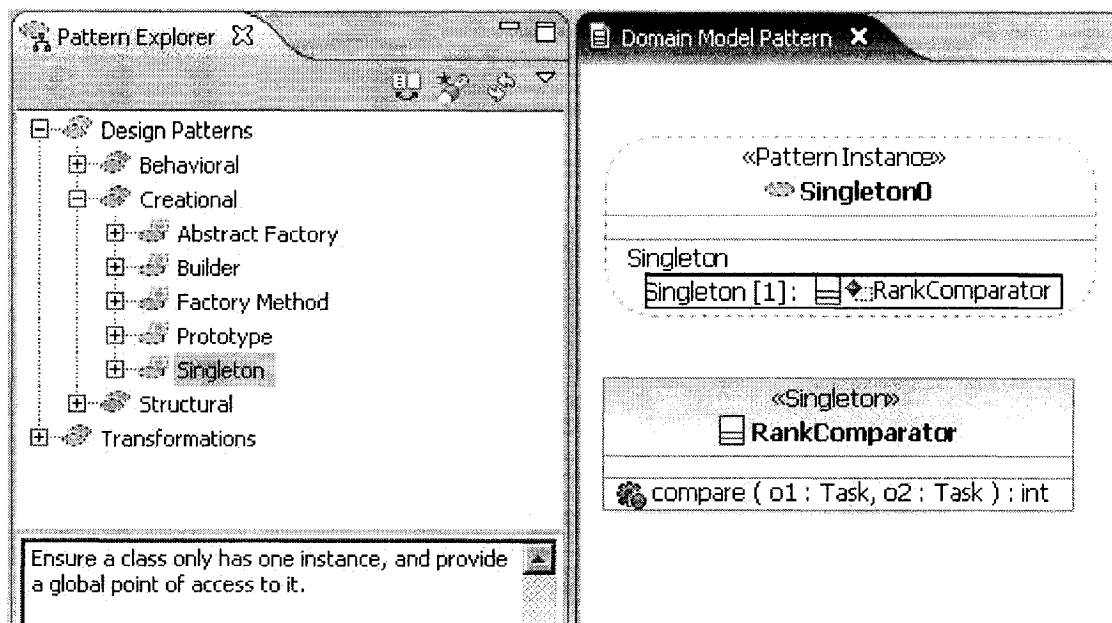
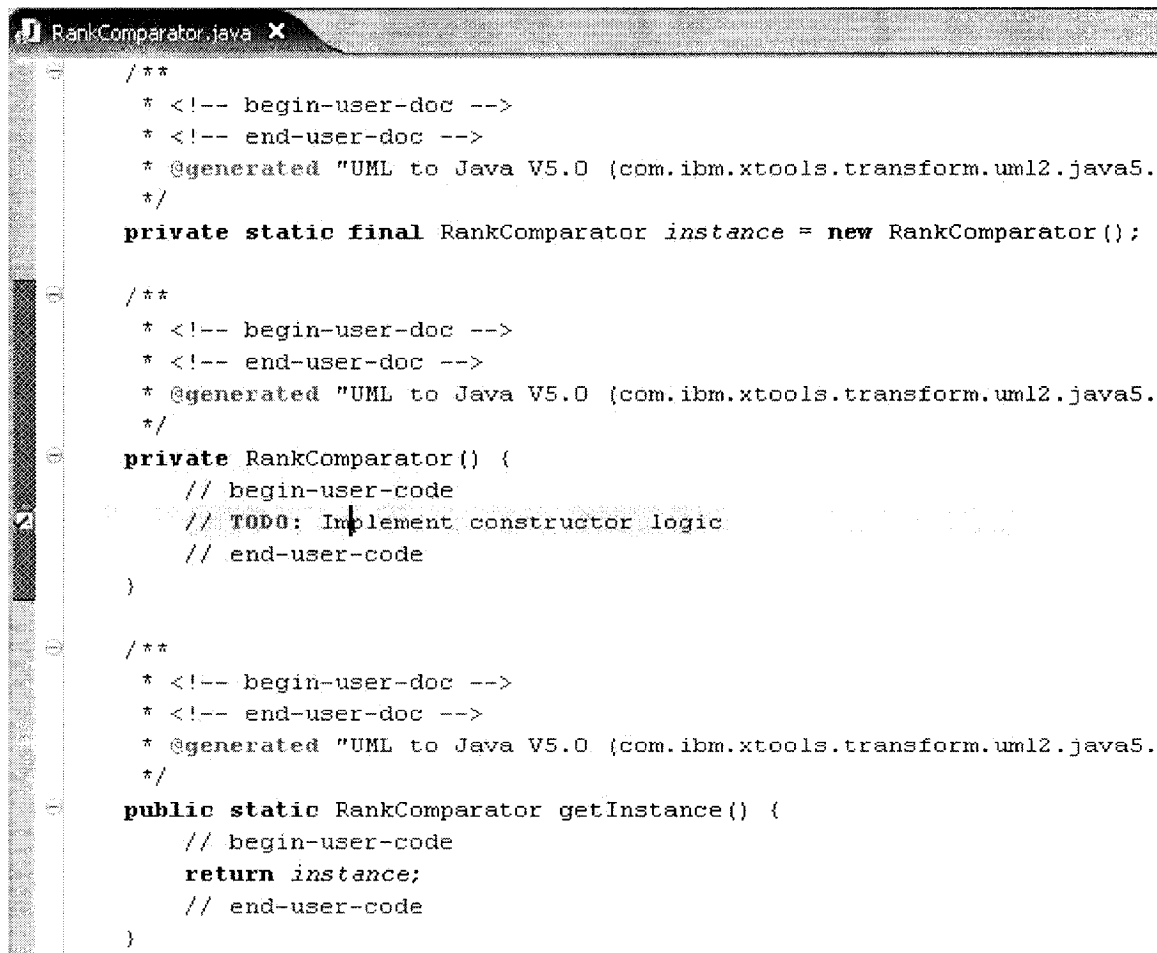


Figure 14: RSA support for Design Patterns



```

RankComparator.java
/**
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated "UML to Java V5.0 (com.ibm.xtools.transform.uml2.java5.
 */
private static final RankComparator instance = new RankComparator();

/**
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated "UML to Java V5.0 (com.ibm.xtools.transform.uml2.java5.
 */
private RankComparator() {
    // begin-user-code
    // TODO: Implement constructor logic
    // end-user-code
}

/**
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated "UML to Java V5.0 (com.ibm.xtools.transform.uml2.java5.
 */
public static RankComparator getInstance() {
    // begin-user-code
    return instance;
    // end-user-code
}

```

Figure 15: Singleton code automatically generated by RSA

4.3.2 Basic Features for Tasks and Folders

In this section, we describe the development of the basic features of creation/addition, editing, and removal of tasks and folders. From the domain model, we see that tasks are always contained in a folder. In addition, since tasks cannot be shared or moved, we do not provide a mutator for the `containingFolder`.

In terms of responsibility assignment, we ask ourselves which class should be responsible for creating `Tasks`. On the one hand, we can simply have a public `Task` constructor, ensuring that a containing folder is provided. On the other hand, the Creator pattern [Larman 2004] advises that the creation of an entity *A* could be assigned to an entity *B* if *B* aggregates *A*. Since a folder aggregates tasks, we could provide a method `newTask()` in the `Folder` class which creates the task and adds it to the folder's task collection.

Considering both options, having a public `Task` constructor is not as simple as it seems. We would also require a `Folder.addTask(task:Task)` method. Before the call to `addTask()`, a newly created task would fail to satisfy the global invariant that all tasks be contained within a folder. Furthermore, this design alternative would actually allow a task to be added to the collections of multiple folders. Such task sharing is not part of our current requirements. Using the Creator pattern is a better choice for the present, as it is a simpler design and it encapsulates task creation, helping us to maintain necessary invariants. Note that under this design, we declare the `Task` constructor to have package scope (Figure 16). The `Task` constructor also assigns rank using its `NEXT_RANK` property. For the removal of tasks, the folder that contains the task removes it from the folder's task collection with a `removeTask(task:Task)` method.

The rationale for folder creation is similar to that for creating tasks. We provide a method `Folder.newFolder()`. However, the creation of root folders varies. A root folder has no containing folder. Hence, the responsibility to create it cannot be assigned to any particular folder. We indicate a folder is a root folder with an `isRootFolder():boolean` method. We can create a root folder with the help of the `Folder` constructor by passing null for the containing folder. The constructor documentation could make it clear that this would result in the creation of a root folder. On the other hand, we could provide a “creation method” [Kerievsky 2004, p.57] named `newRootFolder()` that clearly specifies the kind of folder it returns. Such a self-documenting method, in the opinion of the author, better clarifies the intent of the method while the constructor would need additional documentation to indicate that it is creating a root folder. For the removal of folders, the folder that contains the given folder removes it from its collection with a `removeFolder()` method (Figure 16).

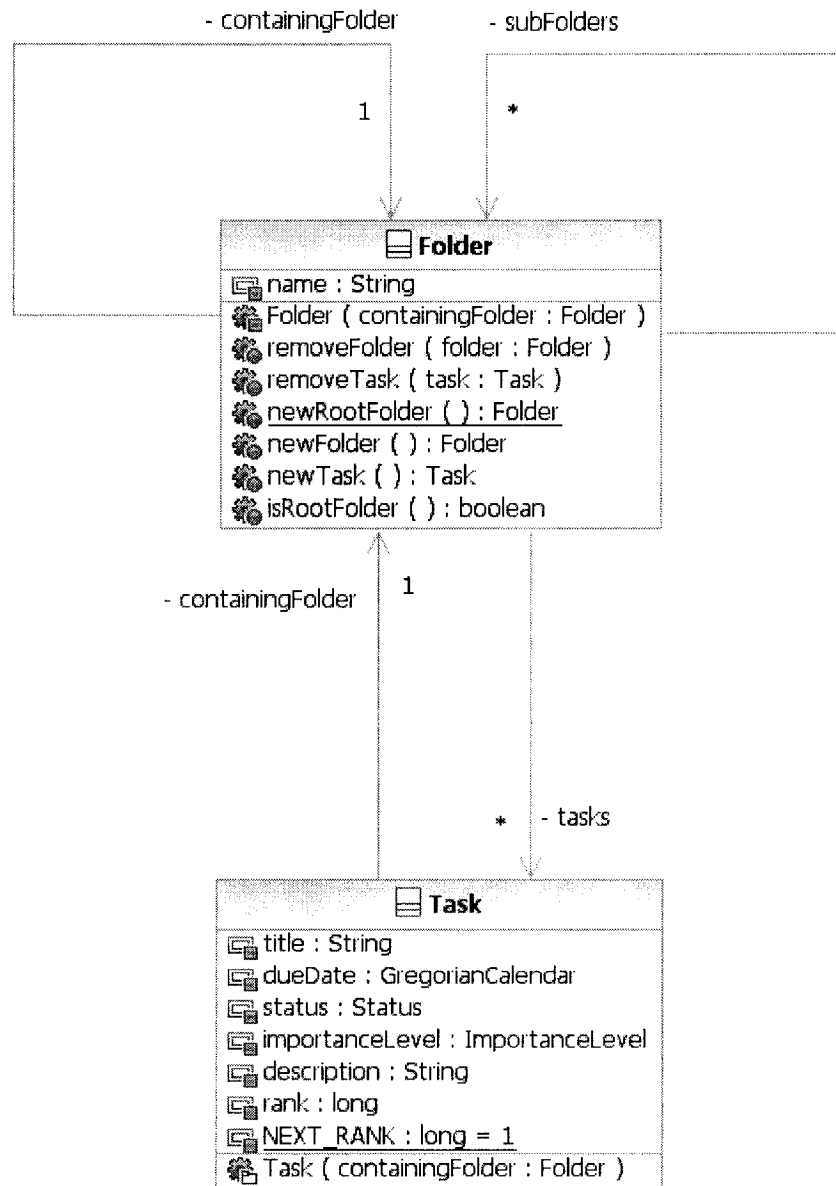


Figure 16: Class Diagram for Basic Features for Task and Folder

After the model is created, we use it to generate code. An example of the modeling-implementation workflow is given below where the model whose view is shown in Figure 16 is forward transformed to a Java project. The forward transformation by RSA creates empty method stubs in the Java project with `@generated` tags and `//begin-user-code` and `//end-user-code` comments within the method body (Figure 17). We then fill the details of the method body between the `//begin-user-code` and `//end-user-code` comments so that our code is not deleted in subsequent forward transformations (Figure 18).

```

/**
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @return
 * @generated "UML to Java V5.0 (com.ibm.xtools.transform.1
 */
public Folder newFolder() {
    // begin-user-code
    // TODO Auto-generated method stub
    return null;
    // end-user-code
}

```

Figure 17: Empty Method body for `Folder.newFolder()` generated by RSA after Forward Transformation

```

/**
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @return
 * @generated "UML to Java V5.0 (com.ibm.xtools.transform.uml2.1
 */
public Folder newFolder() {
    // begin-user-code
    Folder folder = new Folder(this);
    subFolders.add(folder);
    return folder;
    // end-user-code
}

```

Figure 18: Programmer Filled Method body for `Folder.newFolder()` after Forward Transformation

Regarding getters and setters for class properties, there is an option in the transformation configuration file, by which RSA can automatically create getters and setters for the required fields. However, we create the getters and setters as explained in Section 5.6.1.

Note that in Figure 16, the directed associations between `Task` and `Folder` replace the bidirectional association and the composite aggregation shown in the domain model of Figure 11. This is because when RSA performs the reverse transformation from code to model, it merges a temporary model generated from the code and the original UML model. In this merge, the properties in code are considered different from those selected in Figure 11, though the qualifiers of these attributes have not been changed in code. If we

chose to maintain such information in the model, then we manually have to merge them every time we choose to reverse transform code to model.

4.4 Iteration 2: Preliminary Thick Client UI

In previous iteration, we realized the basic functionality for folders and tasks. Next, we create a thick-client UI that provides access to this basic functionality. Specifically, the UI features that are going to be implemented in this iteration are:

- View tasks.
- Add and remove a task from a folder.
- Edit task details.
- View folders.
- Add and remove a non-root folders.
- Edit a folder's name.

These capabilities are realized using the Eclipse Rich Client Platform (RCP) [McAffer 2005]. We choose Eclipse RCP for the following reasons:

- It has a rich source of UI components and forms that can accelerate the development of rich client applications, once the initial learning curve is overcome.
- The widgets created in Eclipse RCP have a look and feel similar to that provided by the underlying operating system.
- For building UIs, the author has more prior experience with Eclipse RCP than with other UI toolkits.

4.4.1 Eclipse Rich Client Platform (RCP)

Eclipse RCP is a subset of the Eclipse Platform providing the minimum set of components required to create rich client applications. The highlighted components in Figure 19 comprise the Eclipse RCP where the upper components build on the lower components [Eclipse RCP].

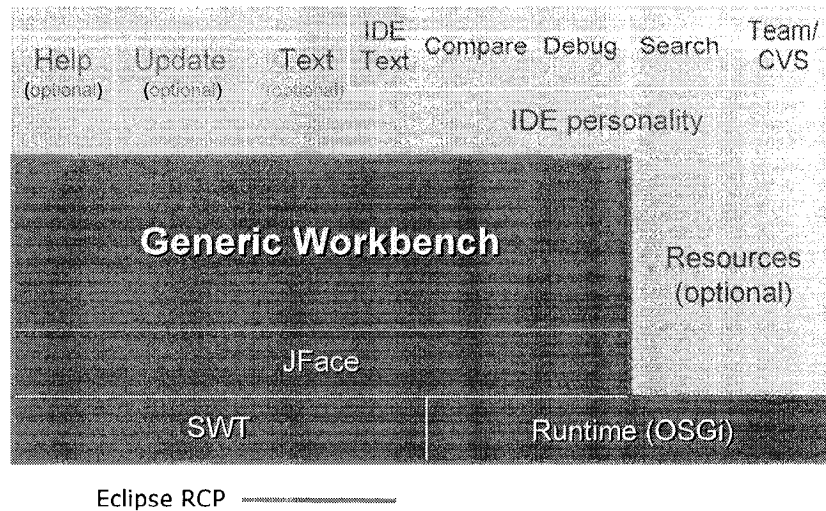


Figure 19: Eclipse Platform Architecture

As seen in Figure 19, one of the basic components in the Eclipse architecture is the Standard Widgets Toolkit (SWT). The SWT is a low-level graphics library providing a thin layer on top of operating system facilities. It provides standard UI controls such as lists, menus, fonts and colors through a consistent, portable Java API [McAffer 2005].

However, SWT is too low level for most UI programming tasks. Eclipse provides a UI toolkit called *JFace* that builds on the SWT in order to shield the developer from low level UI programming. The developer can concentrate on the domain and use JFace to communicate with the UI widgets. One of the ways JFace bridges this gap between domain objects and UI widgets is through *viewers*. Viewers are UI structures that populate the SWT widgets with the required information from domain objects and at the same time, update the UI widgets if there are any changes in the domain objects. In other words, the viewer is the subscriber and the domain object is the publisher [Buschmann 1996].

Additional presentation and coordination to JFace components is provided by the *Generic Workbench*. When a user interacts with the application, he or she may use *views* provided by the Workbench. A *view* is typically used to display and navigate a hierarchy of information, open an editor or display properties for an active editor. The viewer described above normally provides this information to a view. Depending on the viewer, a view can display information as a tree hierarchy or even in a tabular form. Views can also contain other standard UI features such as toolbars and menus. In this section, we use the term view to refer to

Eclipse views. Thus, the JFace toolkit along with the Workbench provides a convenient framework to create rich clients [McAffer 2005].

4.4.2 Feature: Basic UI features for Tasks

As explained in the previous section, the JFace UI toolkit aims to allow developers to focus on the domain though the toolkit requires that a domain object notify its viewers when it changes. This requirement can be implemented in different ways. We can maintain the notifications in each domain object, or we can assign the notification responsibility to a super class of all domain objects. Another option is to create a level of indirection by using UI specific wrapper classes for domain objects.

We choose to keep the domain objects independent of such notification responsibility. Creating a domain super class can be considered when the domain model becomes more mature. For now, we choose to create wrapper classes. This creates maintenance overhead but it serves the purpose of keeping the domain objects independent of notification responsibility. With this basic design decision taken, we move forward and implement the UI features.

The users would like to see the tasks of a single folder or multiple folders collectively. The ideal representation of task information would be a table view to show the important or most relevant aspects of each task at a glance and some view to show the details of each task. Since the table view is a necessary feature, we can use it now to show all relevant details. When the domain becomes more complex, we can add another view to the application. Thus for now, the table view represents the task list of a folder—see Figure 21. In keeping with the design decision to create wrapper classes for the domain classes, we create a wrapper class `FolderTaskListUIWrapper`, which provides access to the folder's task list and notify the UI (UI classes that wish to receive notification have to implement the `ITaskListViewer` interface) when the task domain objects change (see Figure 20).

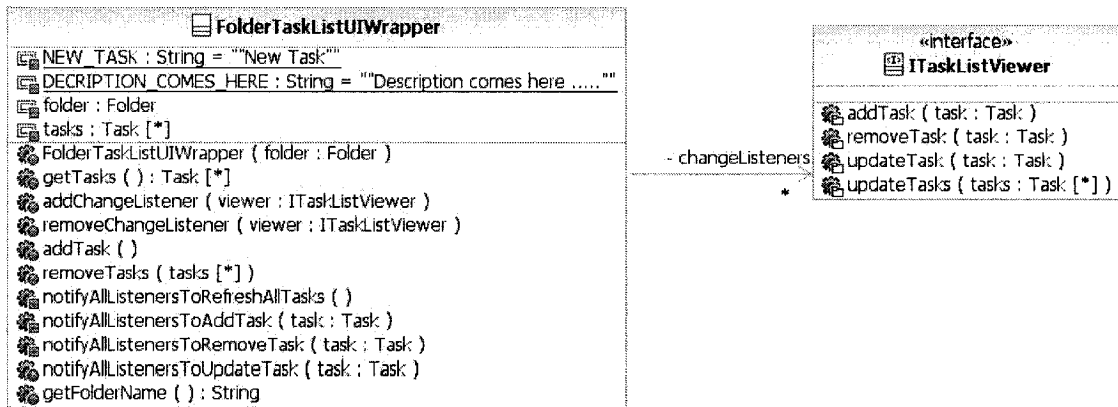




Figure 20: Class diagram for FolderTaskListUIWrapper and ITaskListViewer

I	Title	Due Date(MM/DD/YYYY)	Status	Importance Level	Folder
	OOPSLA 07 deadline	Aug 10 2007	N PROGRESS	A	Research
	Discuss MDD with Asif/Rajiv	Jun 11 2007	NOT STARTED	B	Research
			IN PROGRESS		
			DONE		

Figure 21: Table View of Tasks

As explained above, the task list is represented as a table view with each column representing a task attribute and each row representing a task instance. Therefore, a cell in the table would represent the value of an attribute for a particular task. To create an editable cell, the Eclipse JFace API provides functionality that requires the list of columns in the table view that can be modified, the values to be displayed before editing and the task to be modified. When the user edits a task attribute in the UI, the JFace toolkit changes the state of the corresponding task attribute and the FolderTaskListUIWrapper instance associated with this table view is notified of the change so that it can refresh the table view to show the updated value.

For the addition and deletion of tasks, we create two UI button widgets in a tool bar in the table view—shown as  and , respectively in the top right corner of Figure 21. These widgets call the FolderTaskListUIWrapper instance for selected folder, which updates the folder domain object and notifies the table view to reflect the addition or deletion of tasks.

4.4.3 Feature: Basic UI features for Folders

One of the key aspects for choosing folders to collect tasks is the ability to create folder hierarchies. A hierarchy can be effectively shown in the UI as a familiar tree structure with nodes and leaves—see Figure 22. We

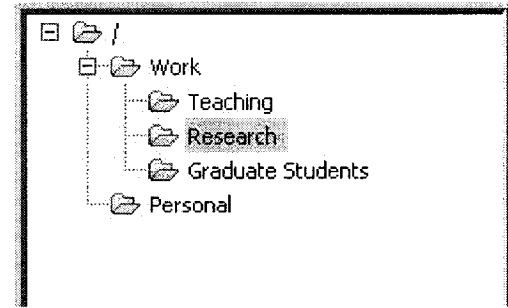


Figure 22: Tree view of folders

can create a class `FolderUIWrapper` (see Figure 23) to

access the folder's list of subfolders. Any addition or deletion of folders is delegated from the `FolderUIWrapper` to the parent folder and UI classes are notified of changes to the folder domain object. UI classes have to subscribe by implementing the `IFolderListViewer` interface shown in Figure 23.

We have chosen to keep the folder tree view as a fixed standalone view on the left (Figure 24). For this reason, we cannot choose the same technique as for tasks for addition and deletion of folders. Therefore, we choose to implement it with the help of a special type of menu called the context menu. A context menu is the menu that is obtained when a user normally clicks the right mouse button in the Windows operating system. It is called a context menu as the actions populated in the menu depend on the entity on which it is clicked. This context menu is filled with actions representing add and delete folders respectively. Therefore, if an action to add folder is clicked in the context menu, a new folder with a default name is

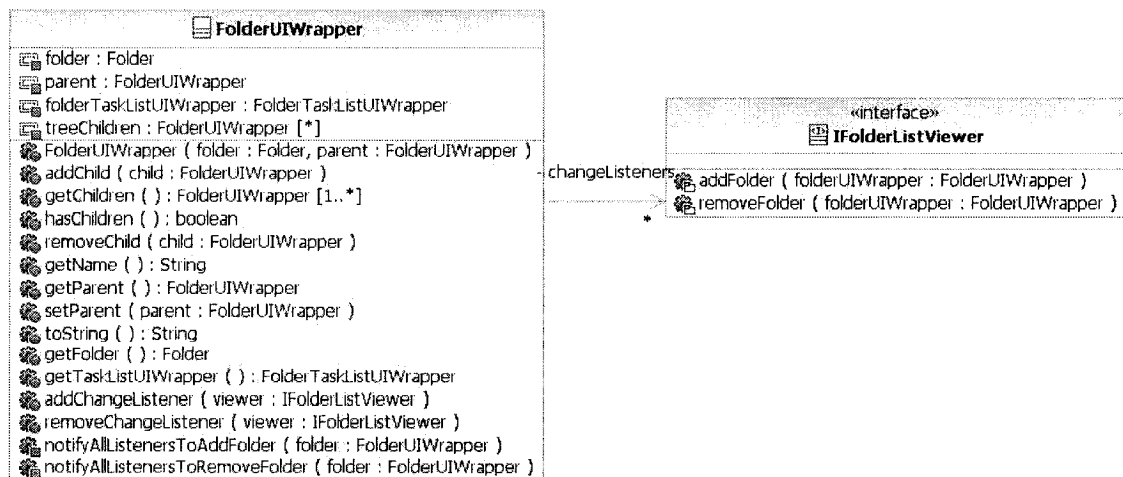


Figure 23: Class diagram for FolderUIWrapper

created and its corresponding `FolderUIWrapper` instance is created.

For editing folder names, we use an implementation very similar to the one used for editing the table view for tasks as mentioned in 4.4.2; i.e., utilize the support provided by JFace where we only have to specify the columns that can be edited, the values to be displayed before editing and the folder whose name is edited.

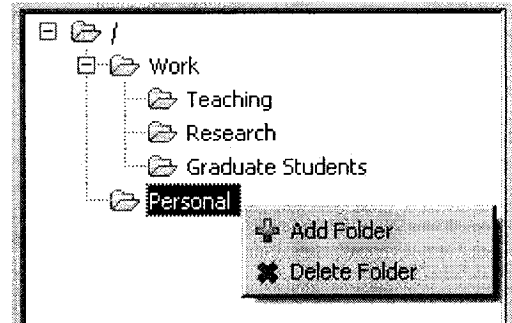


Figure 24: Context menu for Add Folder and Delete Folder

4.5 Iteration 3: View All Tasks for a Folder

In the previous two iterations, we implemented a basic feature set for tasks and folders and created an initial UI for this feature set. In this iteration, we implement the feature of viewing tasks for a particular folder *and* its subfolders.

4.5.1 Feature: View All Tasks of Folder

Users want to see the tasks not only in a specific folder but in its subfolders as well. We provide this feature with a method `Folder.getAllTasks()`. When a folder is requested for its tasks, it will first collect the tasks that reside in it. It will then collect tasks from each of its subfolders, which will do the same, thus calling the `getAllTasks()` recursively. Figure 25 shows the model after adding the above-mentioned feature.

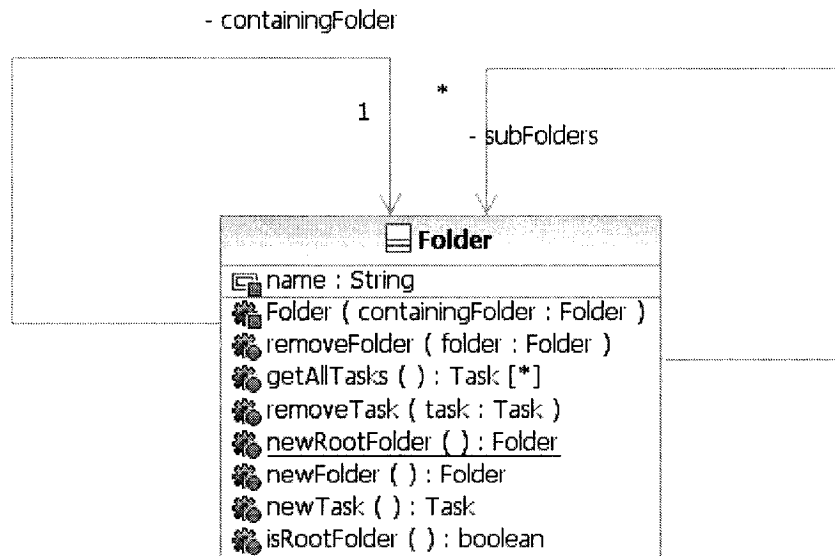


Figure 25: Class diagram after implementing View All Tasks feature

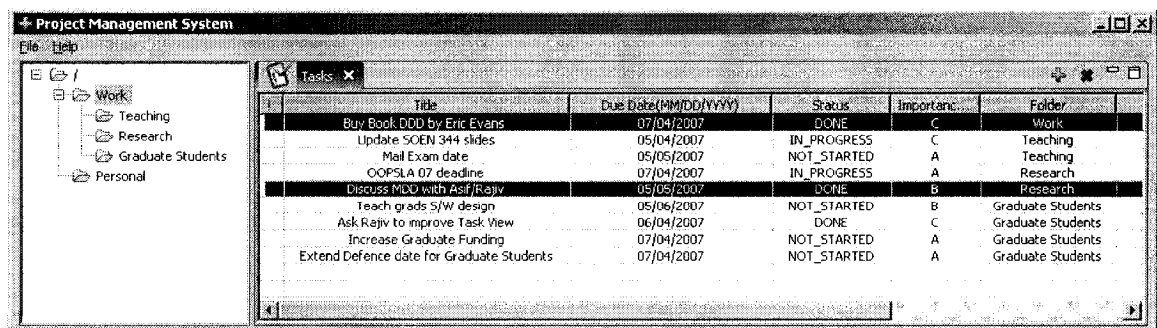


Figure 26: View Tasks by Folder

4.5.2 Feature: View All Tasks by Folder (UI)

In Sections 4.4.2 and 4.4.3, we established a tree view for the folders and a table view for the tasks. We now provide the capability to view tasks by folder in the UI. The Generic Workbench provides a selection service between views. This selection service links views such that a target view can respond to selections in a source view. In our project, we use this selection service so that the table view can show the tasks of a folder selected in the tree view. Figure 26 shows all the tasks in the folder “Work” and its subfolders.

4.5.3 Rework on Remove Task

After implementing the above-mentioned features, a design flaw was discovered. In the UI, when the user selects a folder to see its tasks, we show the tasks not only of the folder but of the subfolders as well. In Figure 26, the user can select a task in folder “Work” and another task in subfolder “Research.” The user would expect that on clicking the delete task button, both these tasks would be deleted. However, the remove task functionality was designed such that the containing folder would do the removal of its tasks. A folder cannot delete the tasks contained in its subfolder. Thus, the task in the subfolder “Research” is not deleted. We solve the problem by asking each task for its containing folder. Each containing folder will then remove its respective task. Since this involved a minor modification, this rework was carried out in this iteration itself.

4.6 Iteration 4: TDD for Sorting Tasks

In this iteration, we explore the use of Test Driven Development (TDD) as a means of implementing the user requirement of being able to sort tasks in a folder by their importance level, due date. We make this early switch to TDD, as one of our secondary goals is to investigate the possibility of combining TDD with MDD. This will allow us to make an initial estimation about its merits and demerits along with the techniques we should use in order to fit it with MDD. Since the implementation and tests for sorting by task importance level and task due date are very similar, we will only describe the development of sorting tasks by importance level. This sorting is to be done in the order of decreasing importance level, i.e., starting from the highest level A and ending with the lowest level C.

4.6.1 Design Rationale

Tasks are stored in a folder according to a user specified order, which we call the ranking of tasks. In this iteration, we will ensure that a sorting functionality will only affect the view of the tasks in the UI and not alter the tasks ranking.

One option for the design is to create a `sort()` method in class `Folder`. We can pass it a `Criterion` by which to compare objects. It can then return a sorted collection without altering the order in the collection `tasks` of the class `Folder`. However, though the operation is placed within the class that has the required

data, sorting is a general responsibility that can be used independently of the criteria for comparison. We may have other domain classes that may want to sort their collections too. Keeping a sort method in each interested class is duplication of code.

In comparison, the Java platform already has support for sorting user-defined objects. In order to make use of this support, the library requires classes that are responsible for comparison to implement the `java.util.Comparator<T>` interface. This interface consists of a method that compares its two arguments. It returns a negative integer, zero or a positive integer if the first argument is less than, equal to or greater than the second argument, respectively. In our application, we create classes `InverseImportanceLevelComparator`, and `DateComparator`. The class `RankComparator` defined earlier allows the user-specified (ranking) order to be shown in the UI. The class `RankComparator` defined earlier allows the user-specified (ranking) order to be shown in the UI. With all these benefits, we choose to use the support provided by Java libraries to sort in our application.

4.6.2 JUnit Tests for Sorting

Thus, in order to sort tasks in decreasing importance level, we first develop the functionality of comparing importance levels. The tests used to develop this functionality are

- **Test 1:** Compare two tasks with the *same* importance level. *Zero* should be returned.
- **Test 2:** Compare two tasks, the first with a *lower* importance level than the second. A *positive* value should be returned.
- **Test 3:** Compare two tasks, the first with a *higher* importance level than the second. A *negative* value should be returned.

In Test 1, we create two tasks with the same importance level. To compare the importance level, we create a class `InverseImportanceLevelComparator`. Figure 27 shows the code for the first test as it is so far. Note that `InverseImportanceLevelComparator` does not yet exist, and hence it is reported as a syntax error. An Eclipse “Quick Fix” for this problem offers to create the class—see Figure 28. The class creation dialog already shows that `InverseImportanceLevelComparator` implements the `java.util.Comparator<T>` interface. When RSA creates the class and its `compare(o1:Task, o2:Task)` method,

it will have a default return value of zero (see Figure 29). One of the TDD principles is that we change the implementation only to fix the failing test [Astels 2003]. Since zero is the value that is expected, we do not change the return value. We add the following final line to `testCompareTasksWithSameImportanceLevel()`:

```
assertEquals("Equal Importance Level not determined",
0, comparator.compare(firstTask, secondTask) );
```

We run the test and it succeeds as expected.

```
31 public void testCompareTasksWithSameImportanceLevel() {
32
33     Folder root = Folder.newRootFolder();
34     Task firstTask = root.newTask();
35     firstTask.setImportanceLevel(ImportanceLevel.B);
36     Task secondTask = root.newTask();
37     secondTask.setImportanceLevel(ImportanceLevel.B);
38     Comparator<Task> comparator = new InverseImportanceLevelComparator();
39
40 }
```

Figure 27: Code showing `InverseImportanceLevelComparator` undefined while writing first test

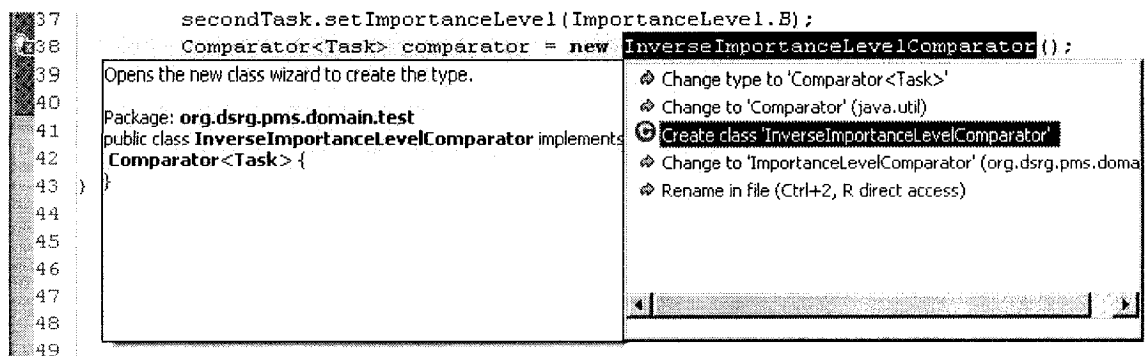


Figure 28: Shows Eclipse Quick Fix feature

```

public class InverseImportanceLevelComparator implements Comparator<Task> {
    public int compare(Task o1, Task o2) {
        // begin-user-code
        return 0;
        // end-user-code
    }
}

```

Figure 29: Shows Class `ImportanceLevelComparator` created by Eclipse with default return value of zero

In Test 2, we create a task having a lower importance level than that of a second task. If we run the two existing tests again, it will fail as the compare method just returns zero, while we need a positive value in this case. Therefore, this test motivates us to modify the implementation. The implementation is enhanced just enough to satisfy Test 2 (see Figure 30). Thus, TDD ensures that all code is tested and that we do not implement features that may not be used at all. We now run the two tests, and they both succeed.

```

public class InverseImportanceLevelComparator implements Comparator<Task> {
    public int compare(Task o1, Task o2) {
        // begin-user-code
        ImportanceLevel firstImpLevel = o1.getImportanceLevel();
        ImportanceLevel secondImpLevel = o2.getImportanceLevel();
        return firstImpLevel.compareTo(secondImpLevel);
        // end-user-code
    }
}

```

Figure 30: compare method for class `ImportanceLevelComparator`

In Test 3, the first task has an importance level higher than that of the second task. Since we have already fully implemented the compare method, all the three tests pass—see Figure 31.

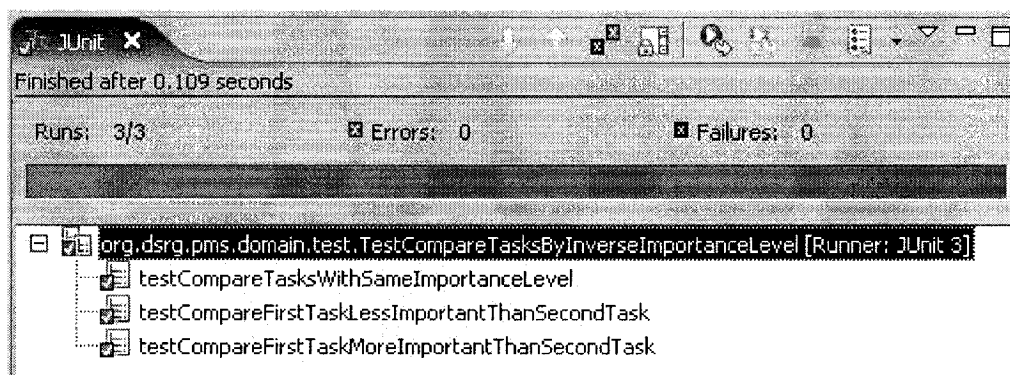


Figure 31: JUnit Eclipse View showing all tests pass indicated by the green bar

With the three tests written, we observe that there is code common to all the three tests: i.e., the creation of an instance of `InverseImportanceLevelComparator`, the first task and the second task. This block of code is refactored to the `setup()` fixture method that is run before every test—see Figure 32. Thus, by

removing duplication, we improve the quality of test code. When such refactorings are performed for an entire suite of tests, it creates more maintainable code.

Now that we have the comparison functionality fully implemented and tested, we should test if the `InverseImportanceLevelComparator` (in conjunction with Java support for sorting) satisfies our requirements.

The test cases we create for the sort functionality are

- **Test 4:** Sort Empty Set
- **Test 5:** Sort Already Ordered Set
- **Test 6:** Sort Unordered Set

The code for these tests is shown in Figure 32 and Figure 33. These tests succeed on execution, signifying that we have correctly utilized the Java support for sorting. Now that the sorting functionality is tested, we follow the same procedure for due date also. We can now begin implementing the UI for the sorting functionality.

```

package org.dsrg.pms.domain.test;

import ...

public class TestCompareTasksByInverseImportanceLevel extends TestCase {

    private Folder root;
    private Task firstTask;
    private Task secondTask;
    private final Comparator<Task> comparator = new
InverseImportanceLevelComparator();

    protected void setUp() throws Exception {
        super.setUp();
        root = Folder.newRootFolder();
        firstTask = root.newTask();
        firstTask.setImportanceLevel(ImportanceLevel.B);
        secondTask = root.newTask();
    }

    public void testCompareTasksWithSameImportanceLevel() {
        secondTask.setImportanceLevel(ImportanceLevel.B);
        assertEquals("Equal Importance Level not determined",
0, comparator.compare(firstTask, secondTask) );
    }

    public void testCompareFirstTaskLessImportantThanSecondTask(){
        secondTask.setImportanceLevel(ImportanceLevel.A);
        int result = comparator.compare(firstTask, secondTask);
        assertTrue("comparison result == " + result + "; it should to be > 0",
            result > 0);
    }

    public void testCompareFirstTaskMoreImportantThanSecondTask() {
        secondTask.setImportanceLevel(ImportanceLevel.C);
        int result = comparator.compare(firstTask, secondTask);
        assertTrue("comparison result == " + result + "; it should to be < 0",
            result < 0);
    }
}

```

Figure 32: Code for TestCompareTasksByInverseImportanceLevel

```

package org.dsrg.pms.domain.test;
import ...

public class TestTaskSortByInverseImportanceLevel extends TestCase {

    private Folder root;
    private final Comparator<Task> comparator = new
InverseImportanceLevelComparator();
    private List<Task> actualTasks;
    private List<Task> expectedTasks;
    private Task taskA1, taskA2, taskB1, taskC1, taskC2;

    protected void setUp() throws Exception {
        super.setUp();
        root = Folder.newRootFolder();
        actualTasks = new ArrayList<Task>();
        expectedTasks = new ArrayList<Task>();
    }

    public void testSortingEmptyList() {
        actualTasks.addAll(root.getAllTasks());
        Collections.sort(actualTasks, comparator);
        compareCollectionByImportanceLevel(expectedTasks, actualTasks);
    }

    public void testSortingAlreadyOrderedList() {
        taskA1 = createTaskInRootFolder("taskA1", ImportanceLevel.A);
        taskA2 = createTaskInRootFolder("taskA2", ImportanceLevel.A);
        taskB1 = createTaskInRootFolder("taskB1", ImportanceLevel.B);
        taskC1 = createTaskInRootFolder("taskC1", ImportanceLevel.C);
        taskC2 = createTaskInRootFolder("taskC2", ImportanceLevel.C);

        expectedTasks.add(taskA1);
        expectedTasks.add(taskA2);
        expectedTasks.add(taskB1);
        expectedTasks.add(taskC1);
        expectedTasks.add(taskC2);

        actualTasks.addAll(root.getAllTasks());
        Collections.sort(actualTasks, comparator);
        compareCollectionByImportanceLevel(expectedTasks, actualTasks);
    }

    public void testUnOrderedList() {
        taskC2 = createTaskInRootFolder("taskC2", ImportanceLevel.C);
        taskC1 = createTaskInRootFolder("taskC1", ImportanceLevel.C);
        taskB1 = createTaskInRootFolder("taskB1", ImportanceLevel.B);
        taskA2 = createTaskInRootFolder("taskA2", ImportanceLevel.A);
        taskA1 = createTaskInRootFolder("taskA1", ImportanceLevel.A);

        expectedTasks.add(taskA1);
        expectedTasks.add(taskA2);
        expectedTasks.add(taskB1);
        expectedTasks.add(taskC1);
        expectedTasks.add(taskC2);

        actualTasks.addAll(root.getAllTasks());
        Collections.sort(actualTasks, comparator);
        compareCollectionByImportanceLevel(expectedTasks, actualTasks);
    }

    private Task createTaskInRootFolder(String taskTitle, ImportanceLevel iLevel) {
        Task task = root.newTask();
        task.setTitle(taskTitle);
        task.setImportanceLevel(iLevel);
        return task;
    }

    private void compareCollectionByImportanceLevel(Collection<Task>
tasks1, Collection<Task> tasks2) {
        Iterator<Task> i1 = tasks1.iterator();
        Iterator<Task> i2 = tasks2.iterator();

        while(i1.hasNext()) {
            Task task1 = i1.next();
            Task task2 = i2.next();
            assertEquals("Importance level ", task1.getImportanceLevel(),
task2.getImportanceLevel());
        }
    }
}

```

Figure 33: Code for TestTaskSortByImportanceLevel

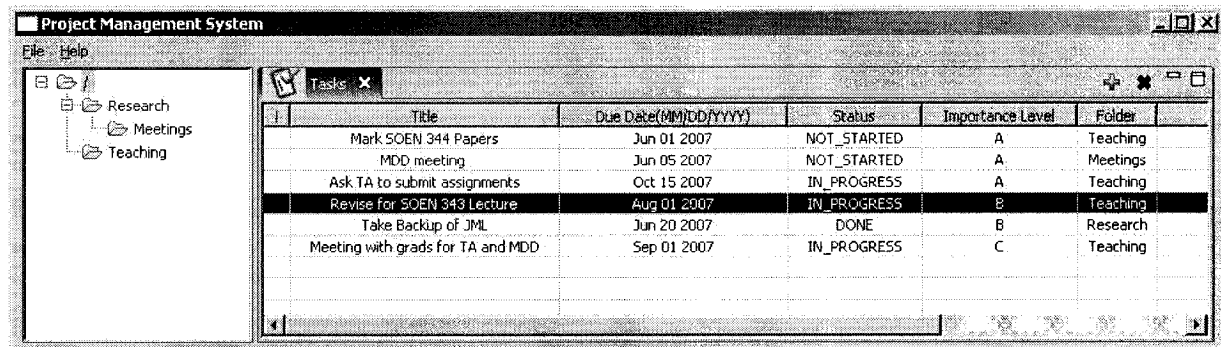


Figure 34: Sorting tasks by column widget

4.6.3 Sorting in UI

For the implementation of sorting in the UI, we can use the column widgets of the table view. Figure 34 shows the table view with many tasks. The column widgets are just above the first task, starting with column widget named “!”. ”.

We can fire an event when any of the three column widgets or *Importance Level* is clicked. Clicking the column widget “!” will sort the tasks in ascending order of rank. The respective column event handler will supply the particular comparator to the current `FolderTaskListUIWrapper` instance shown in the table view. The `FolderTaskListUIWrapper` instance will notify the table view that it has to refresh itself. The view then requests the `FolderTaskListUIWrapper` instance for the task list. The `FolderTaskListUIWrapper` instance sorts the tasks and provides it to the view.

4.6.4 Lessons Learned

In TDD, the developer will write a test and execute it. If the test fails, the developer will then write the implementation to fix that particular test only and no more. If the test succeeds, the developer will write another test and execute it. If the latest test fails, then the developer enhances the code to fix this test and no more. In this manner, the design of the code evolves incrementally with each failing test. Agile Modeling is performed [Astels 2003], but the majority of the design is at the code level i.e., at the same level of abstraction.

In MDD, the design is developed in a model, which is a separate artifact. This model is forward transformed into an implementation and the implementation details are filled out. While filling the

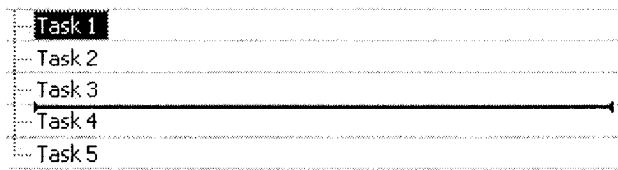


Figure 35: Illustration of task reordering through the UI's drag-and-drop capability: moving Task 1 to after Task 3

implementation details, if the design is found to be incomplete, the design is rectified in the model and again forward transformed into the implementation. In this manner, the design is at the model level i.e., at a higher level of abstraction.

Therefore, both these approaches work at different levels of abstraction to create a design solution. Since our focus is on MDD, we will henceforth design at the model level and will no longer perform TDD. Note that tests will still be written but only after the implementation of each feature.

4.7 Iteration 5: Reordering of Tasks

In this section, we explore the addition of the feature of reordering tasks in a folder (see Figure 35). The user would typically reorder tasks one at a time. This means selecting a task to be reordered and selecting the position to relocate it.

Normally, as observed in previous iterations, we first apply MDD, creating a model and implementation, and then create the UI for the implementation. However, in this case, we know that the Eclipse UI toolkits (SWT and JFace) have a drag and drop (DnD) feature wherein the user can select a task, drag and drop it at a preferred location in the task view. This feature seems well suited for the action of reordering tasks. We do not know how the UI will provide reordering information to the domain layer. The UI could provide the domain layer with indexes of the tasks to be reordered or the actual tasks themselves. For this purpose, we decide to investigate the UI capability before designing the domain layer for this feature.

4.7.1 Initial UI investigation

For the DnD feature, Eclipse SWT introduces the concept of a transfer type. A transfer type is the permissible type of object that can be dragged from a drag source widget and dropped onto another (or the same) widget. Eclipse RCP UI developers can extend the basic transfer type to create domain object

transfer types so that domain objects can also be transferred between widgets. In our case, we would create a `TaskTransfer`, which extends `org.eclipse.swt.dnd.ByteArrayTransfer`, for transferring tasks. The `TaskTransfer` type enables drag and drop by supporting object serialization and deserialization, thus allowing drag and drop across different views or editors. Instead of marshalling the entire `Task` object, we can choose to marshal a unique identifying attribute of the task. This unique identifying attribute will be used to retrieve dragged and dropped tasks from the `TaskTransfer` instance. We will determine this unique identifier during the modeling phase.

The dragged tasks to be serialized are provided by method `dragSetData(event: DragSourceEvent)` of `TaskDragListener` which extends `org.eclipse.swt.dnd.DragSourceAdapter`. Supplementing the SWT support, the JFace toolkit provides an `org.eclipse.jface.viewers.ViewerDropAdapter`, which specifies the action to be taken after the drop. The RCP application developer has to extend this class and perform the desired action, in our case, reordering of tasks. In particular, the action to be done after the drop is validated is carried out in `performDrop(data: Object): boolean` of `TaskTreeDropAdapter` which extends `ViewerDropAdapter` mentioned above.

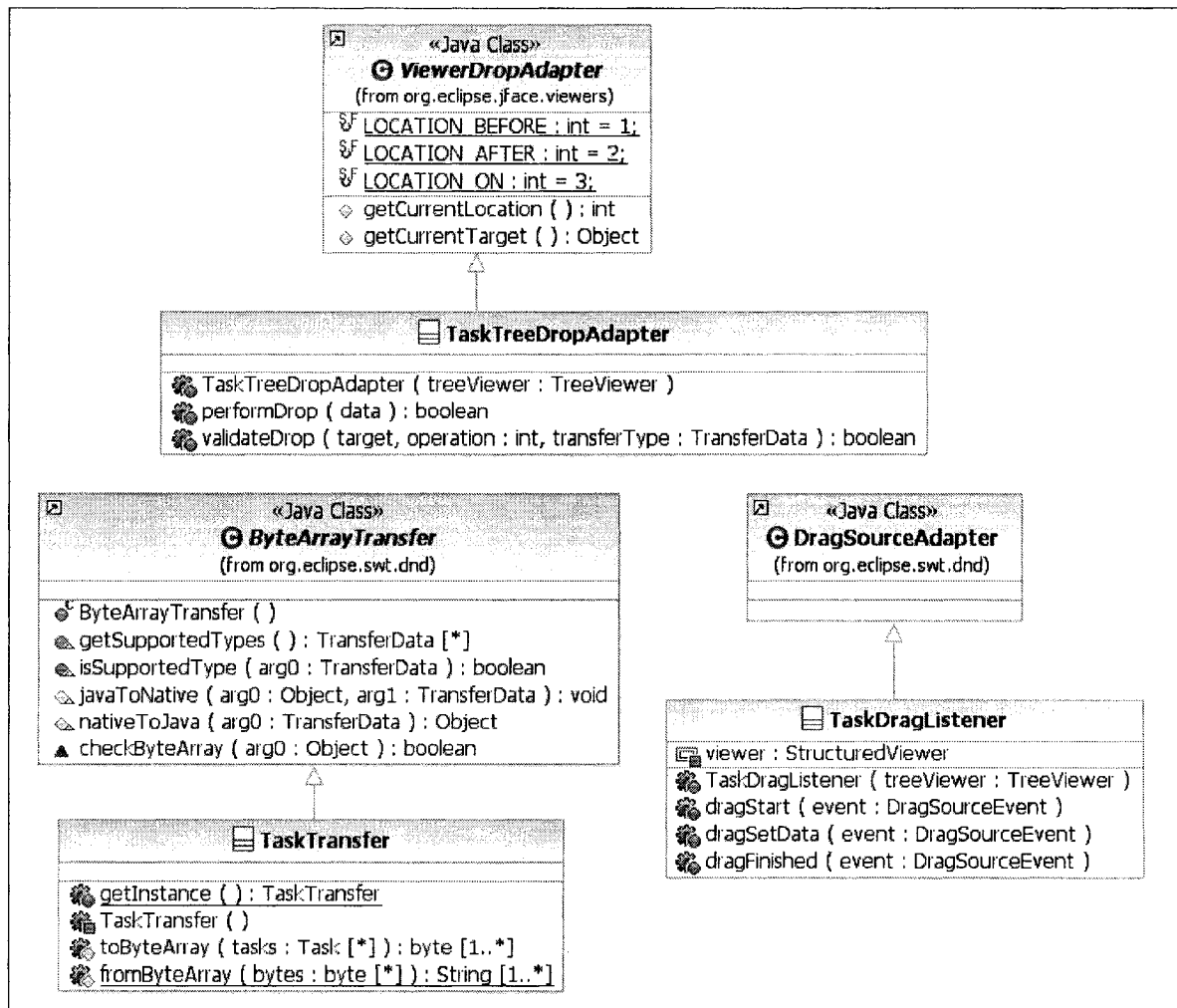


Figure 36: Class Diagram showing UI implementation for DnD feature

JFace provides the following information to the `performDrop()` method:

- the object(s) that are dropped,
- the current target in the drop widget over or near which the drop takes place and
- the location with respect to the current target.

For example, consider the illustration shown in Figure 35. Given tasks Task 1 to Task 5 in ascending order, if we drag-and-drop Task 1 to the location between Task 3 and Task 4 then the dropped object is Task 1, the current target is Task 3 and the location is “after” Task 3. However, if we drag Task 5 to the same place between Task 3 and Task 4, then the dropped object would be Task 5, current target would be Task 4 and the location will be “before” Task 4.

Thus, the location of insertion depends in the direction in which the drag takes place. There are three location positions relative to a drop target task:

- If the drag is made from top to bottom in a table then the drop position is *after* a target task.
- If the drag is made from the bottom to top then the drop position is *before* the target task.
- If the drag is stopped on a task then the drop position is *on* the target task.

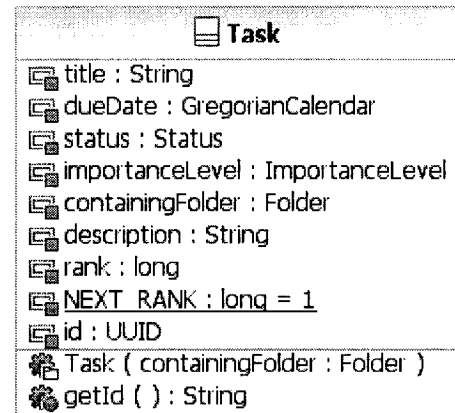


Figure 37: Class Task with UUID

The visual feedback provided by SWT is not supported by the table viewer [Eclipse NewsList 2007] which we have used to display tasks. However, the tree viewer supports visual feedback while at the same time, it can have table like columns. For this purpose, we convert the table view for tasks to a tree view. We manipulate the tree view so that it looks like a simple table of tasks without any hierarchy.

4.7.2 MDD: Unique Task ID

Before starting on the design proper for reordering tasks, we first concern ourselves with finding a unique identifier for tasks so that the dragged task can be retrieved from `TaskTransfer` and provided for reordering. Currently, there is no field in `Task` that can serve to identify a task. Therefore, we have to create a new field `id`, which we assign during the creation of the task. The field `id` can be of type `java.util.UUID`, which is an immutable universally unique identifier (UUID). However, later on, we could possibly use the same `id` to store ids from the database, which may not be of UUID format and value. Therefore, we shall hide the internal storage type of `id` and return it as a `String`. In the future, if the type of `id` changes, the UI code will be protected from such variation. Figure 37 shows the class `Task` with the field `id` and its accessor, which returns a `String` value.

4.7.3 MDD: RankExpert for Reordering Tasks.

Now that we know that the UI can provide the dropped task (using the task `id` field) and the current target task on which it is dropped, we can proceed with designing the domain layer for reordering tasks. In terms

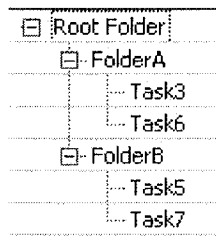


Figure 38: Illustration shows example of class Folder not being a candidate for reordering tasks. Folder A cannot place Task 3 before or after Task 6

of responsibility assignment, we ask ourselves which class should be responsible for reordering tasks. In other words, since we define natural order by rank, which entity should be responsible for reordering tasks by manipulating rank?

The folder seems to be a good choice, as it knows about the tasks it has to reorder. However, the folder knows only about a selection of the entire tasks. There can be two tasks adjacent by rank but residing in different folders – see Figure 38. If a folder A has tasks (ranks are shown as subscript) Task₃, Task₆ and another folder F2 has tasks Task₅, Task₇ then folder A cannot reorder Task₃ after or before Task₆, as it does not know about the tasks before and after Task₆, which are in folder B.

Therefore (by Pure Fabrication [Larman 2004]), we choose to create a new entity `RankExpert` that will be responsible for reordering tasks. Since we now have a `RankExpert`, we can assign it the responsibility of assigning ranks as well. The `RankExpert` will assign rank with the `rank(task:Task)` operation. We could instead choose to return a rank to the task when it is being created, but we want to present a consistent level of abstraction in the `RankExpert` API [McConnell 1993]. Since we have an entity to assign ranks, we should also consider revising the ranks when tasks are deleted. We could revise the ranks of all tasks after each task deletion but this could have a performance overhead. Therefore, we choose to revise all ranks only when the application is opened using the operation `RankExpert.reviseAllRanks()`. However, the `reviseAllRanks()` operation should be called when the application is opened. This is because we should be able to rank newly created tasks after the last rank of the tasks previously persisted. We also realize that a single instance of `RankExpert` for ranking tasks is sufficient. Therefore, we can use the Singleton pattern [Gamma et al. 1995] to create `RankExpert`.

4.7.4 MDD: Reordering Tasks

Once the responsibilities of assigning and revising rank are finalized, we focus on reordering tasks. Considering that we will be provided the reordering information as dropped task and drop target task (the task near which the drop takes place), we could specify an operation `reorder(previousTaskAtThatPosition:Task, newTaskAtThatPosition:Task)` where the drop target task is `previousTaskAtThatPosition` and the dropped task is `newTaskAtThatPosition`. We do not want to use the UI terminology of dropped task and drop target, as we would like to keep the domain layer independent of UI concepts.

Such an operation would suffice for all except one type of reordering, namely reordering any task to the last position of a folder. Note that a folder can show tasks that belong to its subfolders as well. The user may want to reorder a subfolder task to the last position of the parent folder. As there can be a hierarchy of folders, there can be parent and grandparent folders to whose last position we want to reorder the subfolder task. Hence, we will explicitly pass the folder as an argument as well. Since we want the folder as an argument for reordering to the last position only, we create a separate operation `reorderToLastPositionInFolder(folder: Folder, newTaskAtThatPosition: Task)`.

However, for reordering to the last position of a folder, we need to reorder relative to the last task of that folder. Since we need the last task of a folder, we will create an operation `Folder.lastTask():Task`. This operation works for all cases except when the folder is empty. The operation could return the value `null` to signify that there are no tasks in the folder. The author is of the opinion that, instead of overloading `null` with a different interpretation, it is better to create an explicit check to see if the folder has tasks. This could be done with an operation `Folder.hasTasks():boolean`. This requirement that the query be checked before calling `lastTask():Task` is mentioned in the model documentation.

All the above operations of `RankExpert` have been made with the assumption that the total set of tasks in the application will be provided to `RankExpert`. How do we provide these tasks to the `RankExpert`? We could provide the root folder to the `RankExpert`, or we could just assign the tasks with the help of a setter. We will decouple the folder from the `RankExpert` and provide the total set of tasks via a setter method

`setTasks(tasks: Task[*])`. This setter method should be called before each of the `RankExpert` operations so that `RankExpert` has the latest collection of tasks. This requirement is documented.

After the `RankExpert` manipulates the ranks of tasks, the tasks will remain in their previous positions in `Folder.tasks`, irrespective of their newly updated ranks. We have to restore `Folder.tasks` to contain tasks in their natural ranked order.

The decision to be taken is, “who notifies the folder to restore the order of its internal tasks?” Since the `RankExpert` is the entity that changes the rank, we can ask it to inform the task’s containing folder if it changes a task rank. We could enforce this by creating a private method which changes the rank as well as notifies the containing folder. We could also ensure that the task’s containing folder is notified in `Task.setRank(rank:long)`. Alternatively, we could restore the natural order when the tasks are requested using `Folder.getAllTasks()`.

Restoring the folder order in `RankExpert` is open to error, as it may be forgotten to notify the task’s containing folder every time the task’s rank is changed. Setting the rank in the task setter makes the task aware that the folder collects tasks by rank. In addition, setters and getters should ideally just get and set a value, respectively, and nothing else [Fowler 2003, p. 45]. This could complicate the design later. Restoring the order in `Folder.getAllTasks()` would create an overhead of ensuring that the tasks are always in natural order every time `Folder.getAllTasks()` is called, even though the tasks may not be reordered at all.

Thus we have a choice between either introducing a chance of error in programming or complicating the design and increasing performance overhead. In this case, we choose to have a slight increase in performance overhead and provide a private method `Folder.restoreNaturalTaskOrder()` which will be called by `Folder.getAllTasks()`.

4.7.5 Providing the Right Tasks to the RankExpert

Once the `RankExpert` class operations have been added to the model, we need to decide how to provide the correct tasks to it from the UI. As explained previously, there are three different types of locations relative to the target task (i.e., the task on or adjacent to which another task is dropped). The `RankExpert`

can directly be used when the location is on or before the target task as we can pass the target task as `previousTaskAtThatPosition` in `reorder(previousTaskAtThatPosition, newTaskAtThatPosition)`.

However, when the location of drop is after the target task, we have to consider the task next to the target task and provide the next task as the argument for `previousTaskAtThatPosition` (Figure 39). Note that the next task in this case is the next task in all the tasks of a folder (including the tasks of its subfolder). It is not the next ranked task, which may reside in another folder.

Since the next task is relative to the folder, the folder can provide the next task with `taskNextTo(task: Task): Task`. However, this operation will return a `null` value if the argument parameter `task` is the last task of the folder. We can adopt the convention that the UI is to treat a `null` return value, as an indication that `task` is the last task of the folder. However, if `task` is not contained in the folder, the operation will return `null` as well. As can be seen, the same `null` value is overloaded with different interpretations. The author therefore prefers to minimize the use of `null` by creating an explicit query `hasTaskNextTo(task: Task): boolean`. Since `hasTaskNextTo()` will return `false` if `task` is the last task in the folder, we can use it to call the `reorderToLastPositionInFolder(folder: Folder, newTaskAtThatPosition: Task)`. Figure 40 shows the sequence diagram explaining how reordering takes place with the next task or to the last position in folder when `TaskTreeDropAdapter.reorderWithTaskNextTo()` is invoked. Figure 41 and Figure 42 show the class diagram and UI for the feature reordering tasks, respectively.

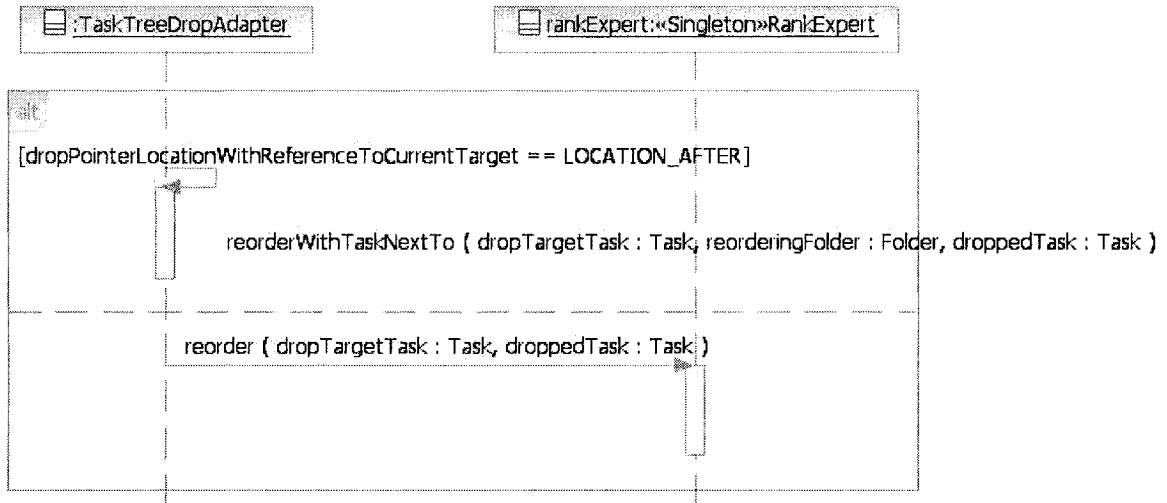


Figure 39: Sequence Diagram showing TaskTreeDropAdapter checking drop location

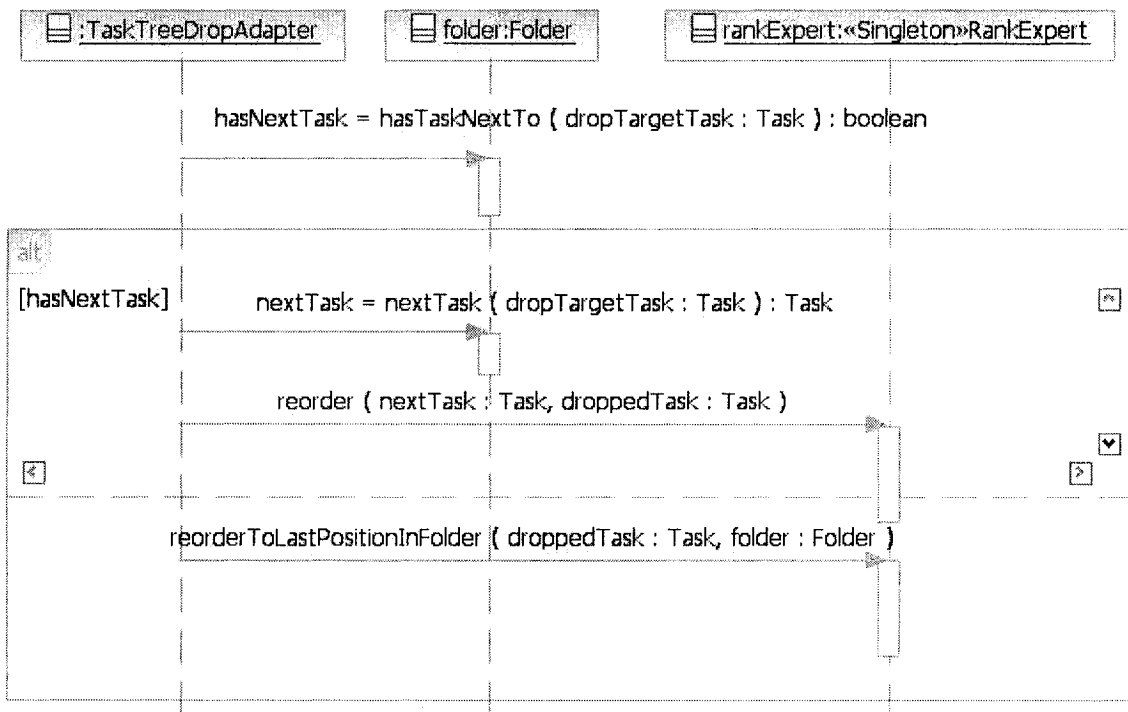


Figure 40: Sequence Diagram showing TaskTreeDropAdapter reordering with next task

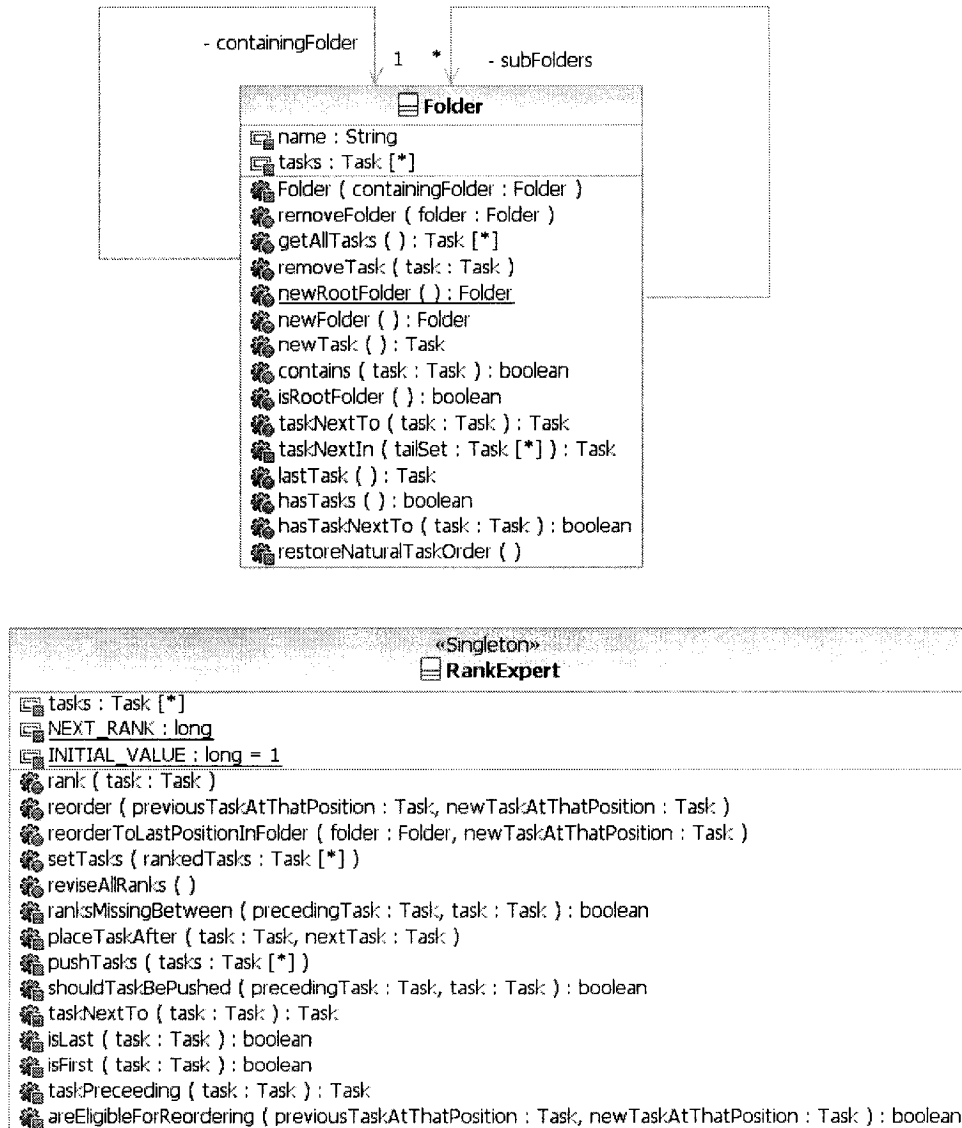


Figure 41: Class Diagram for Reordering Tasks feature

Figure 43 shows the high-level class diagram of the domain layer to the current point of development. Note that `DateComparator`, `InverseImportanceLevelComparator` and `RankComparator` were created directly in code and are obtained by reverse transforming using RSA. At this point, RSA does not support implementing parameterized interfaces. So during reverse transformation, RSA did not reverse transform the fact that all comparators implement `java.util.Comparator<Task>`.

Tasks					
	Title	Due Date(MM/DD/YYYY)	Status	Importanc...	Fc
	Task 1	Jun 18 2007	NOT_STARTED	C	Per
	Task 2	Jun 18 2007	NOT_STARTED	C	Per
	Task 3	Jun 18 2007	NOT_STARTED	C	Per
	Task 4	Mar 19 2007	NOT_STARTED	C	Per
	Task 5	Jun 18 2007	NOT_STARTED	C	Per

Figure 42: Task View showing Drag and Drop feature

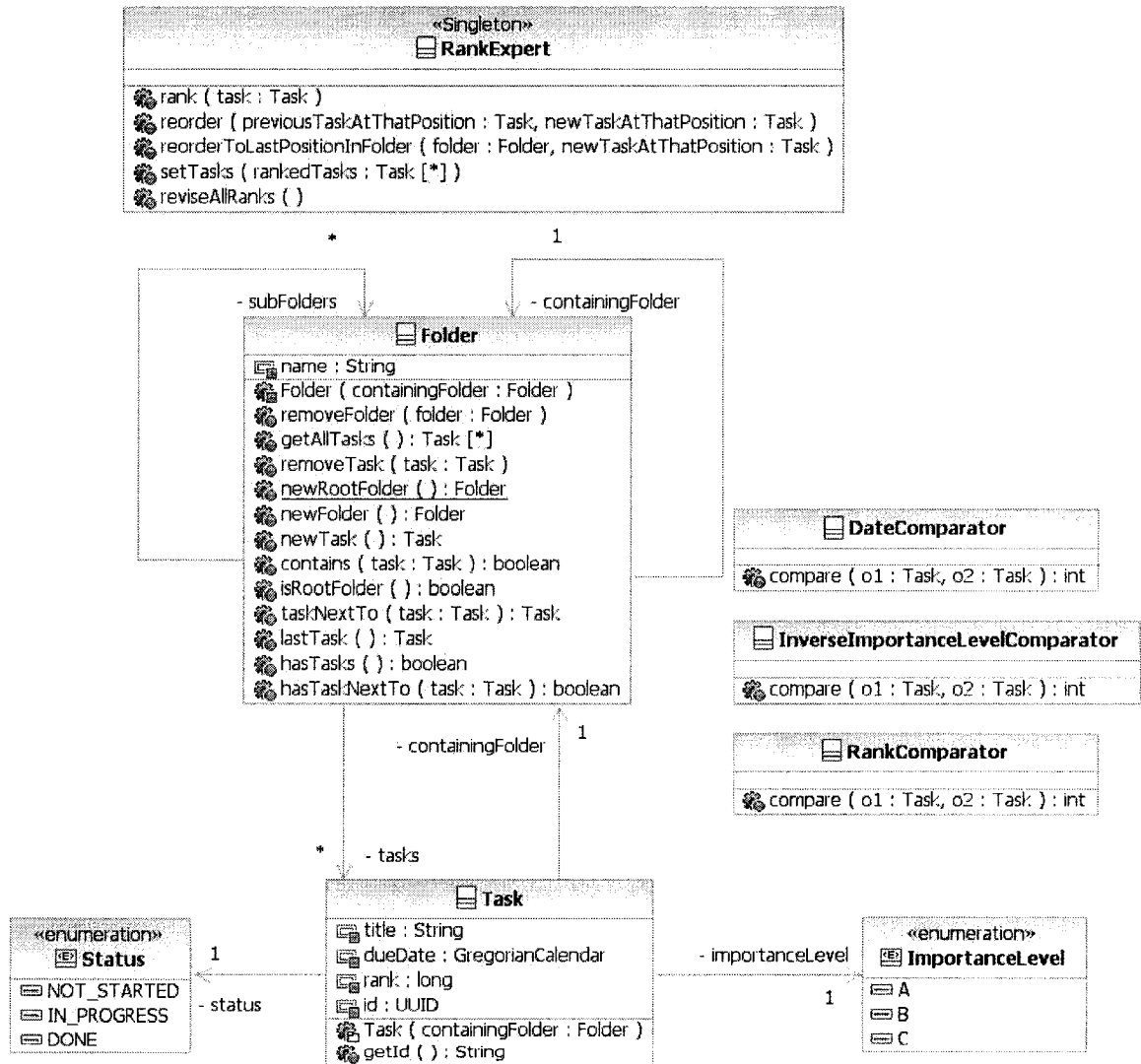


Figure 43: Class Diagram showing Domain Layer after implementing reorder feature

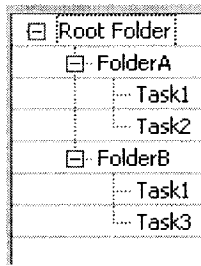


Figure 44: Why a folder cannot remove a subfolder's tasks

4.8 Iteration 6: Share Tasks

4.8.1 Domain Analysis

In this iteration, the feature of sharing tasks between folders is implemented. In discussions with the users, other concepts of file sharing were considered. The usage of symbolic links (a metaphor used in the UNIX operating system) was contemplated, which would make a distinction between the folder containing the task and the folder containing the link. Another option was just to increase the multiplicity of containing folders so that each containing folder had equal rights over the shared task. For the Project Management System, it was decided that making a distinction between a containing folder and a sharing folder did not add much value or flexibility as compared to having multiple containing folders. It was decided that at the current stage of development, it was not necessary to add another domain concept. Thus, the users agreed that a task was shared if it had multiple containing folders.

Since a task could now have multiple containing folders, a user could choose to remove a task in two different ways; the task could only be removed from the selected containing folder or the task could be removed entirely from the application. Both these concepts should be considered while designing.

Once we settled on having multiple containing folders, as in most cases, we also have to consider the impact of being able to view all the tasks in a folder including its subfolders' tasks as well. Consider the case shown in Figure 44, in which *Task1* is shared by both *Folder A* and *Folder B*. Since they are subfolders of *Root Folder*, *Task1* will also be visible if the user chooses to view all tasks of *Root Folder*. If the user removed *Task1* from the view of *Root Folder*, the subfolder to remove it from is not clear. Hence, we keep the restriction that a task can only be removed by its containing folder.

4.8.2 Feature: Sharing Tasks

First, the multiplicity of containing folders is increased to be greater than one. Therefore, `Task` will have the methods `addContainingFolder(folder:Folder)` and `removeContainingFolder(folder:Folder)`. We need to know if a task is shared in order to activate the context menu. By definition, a task is shared if the folder is a containing folder and if the task has multiple containing folders. Both these pieces of information reside in `Task`. Therefore, by the principle of *Information Expert* [Larman 2004], we create a method `Task.isSharedBy(folder:Folder)`. Since a task can now belong to multiple folders, we introduce a method `Folder.addTask(task:Task)`. When a folder adds a task, it will add itself to the task's set of containing folders. Likewise, when a folder removes a task, it will remove itself from the task's containing folders. For deleting a task from the application, the task is removed from each of its containing folders. The class diagram for the share task feature is shown in Figure 45.

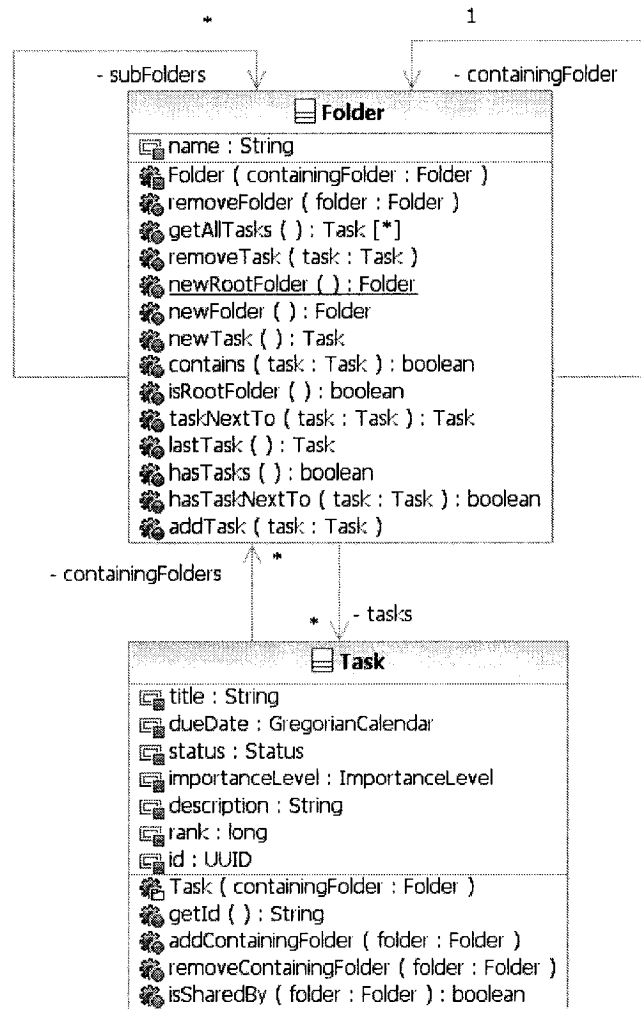


Figure 45: Class Diagram after adding Share Task feature

4.8.3 Sharing Tasks in UI

The Eclipse UI feature of drag and drop can be reused for sharing tasks as well. Note that the same drag and drop functionality is being used for two separate views. For reordering of tasks in the same folder, tasks are dragged and dropped on the same task view. However, for sharing of tasks between folders, the tasks are dragged and dropped on to the folder view. In Figure 46, a task *Meeting with grads for TA and MDD* of folder *Research* is dragged and dropped into the folder *Teaching*. To implement this feature, we create a `FolderTreeDropAdapter` that handles the sharing of tasks after a task is dropped onto the folder view. It is similar to `TaskTreeDropAdapter` created in the feature for reordering tasks.

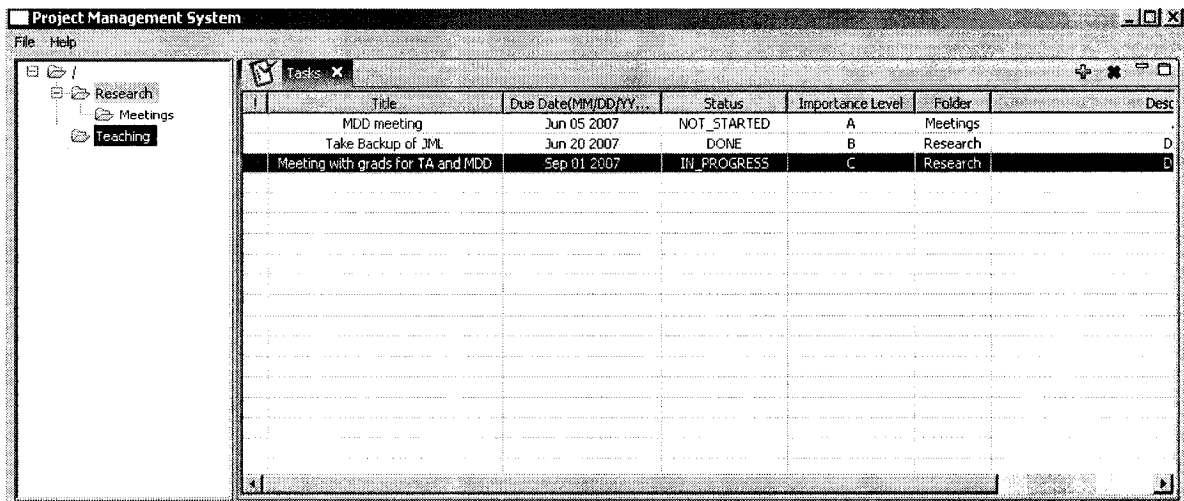


Figure 46: Snapshot of UI showing a task dragged and dropped for sharing between folders

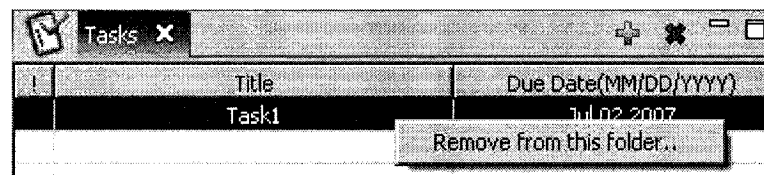


Figure 47: Snapshot of Context Menu for Removing shared Tasks

Whether a task is shared is a property of that task. Therefore, the UI action of removing a task from a particular folder only (and not from the application) should be placed in a context menu (see Figure 47) for each task.

If multiple tasks are selected, the context menu should be inactive even if one of the selected tasks is not a shared task. The responsibility for performing such a check is done by `FolderTaskListUIWrapper` with the method `areAllTasksSharedByFolder(tasks:Task[*])`.

4.8.4 Refactoring

While creating a context menu for the task view, it is observed that there would be significant duplication of code in the task view and the folder view (the folder view uses a context menu to add and remove folders). Applying the refactoring *Extract Class* [Fowler 2000], we move the context menu related code into a class `ContextMenu`. Figure 48 shows the `FolderView` before the refactoring with the relevant context menu code. Figure 49 shows the extracted class `ContextMenu` with the updated `FolderView`.

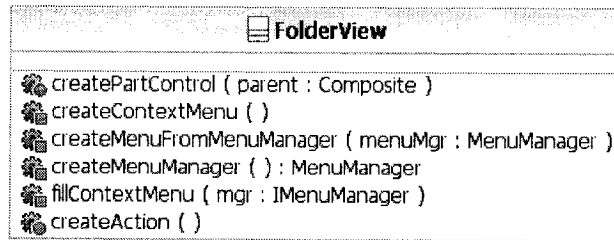


Figure 48: Class Diagram of FolderView before refactoring

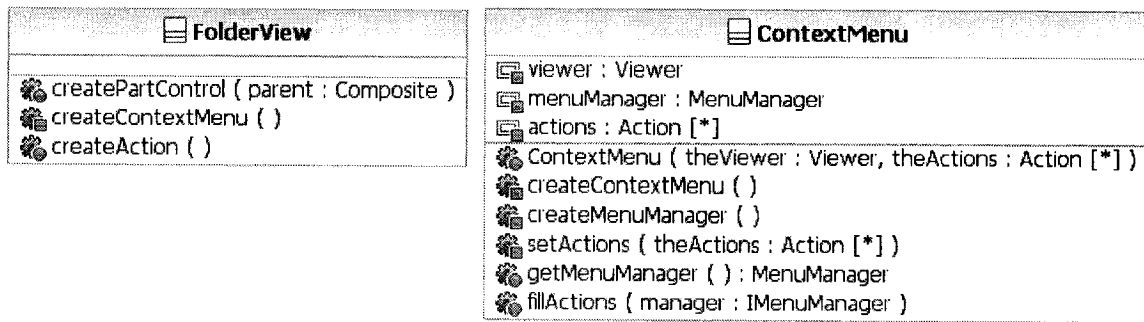


Figure 49: Class Diagram of FolderView and ContextMenu after *Extract Class*

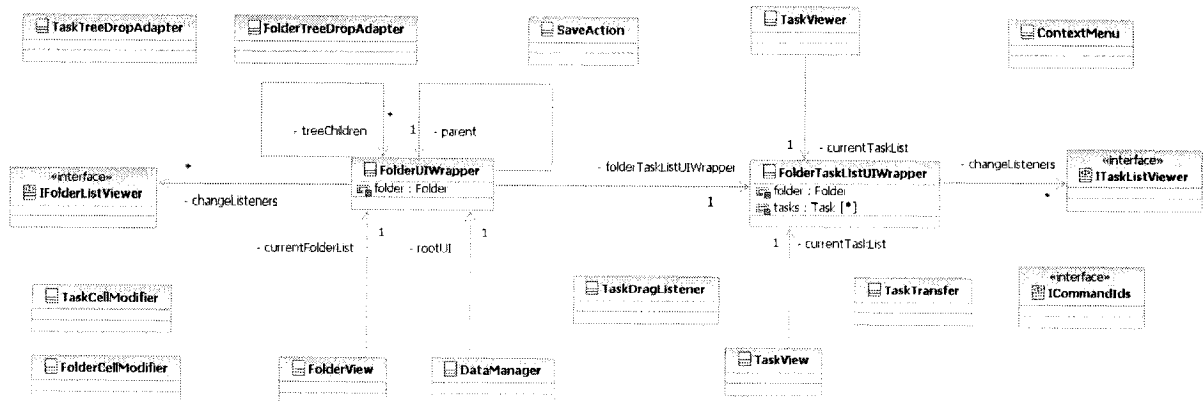


Figure 50: Class Diagram for Presentation Layer

4.8.5 Class Diagram of PMS

At this point, a filtered view of all the classes in the Presentation layer (see Figure 50) and the domain layer (see Figure 51) is shown. Note that the presentation layer does not show the RCP specific classes which are created with every RCP application.

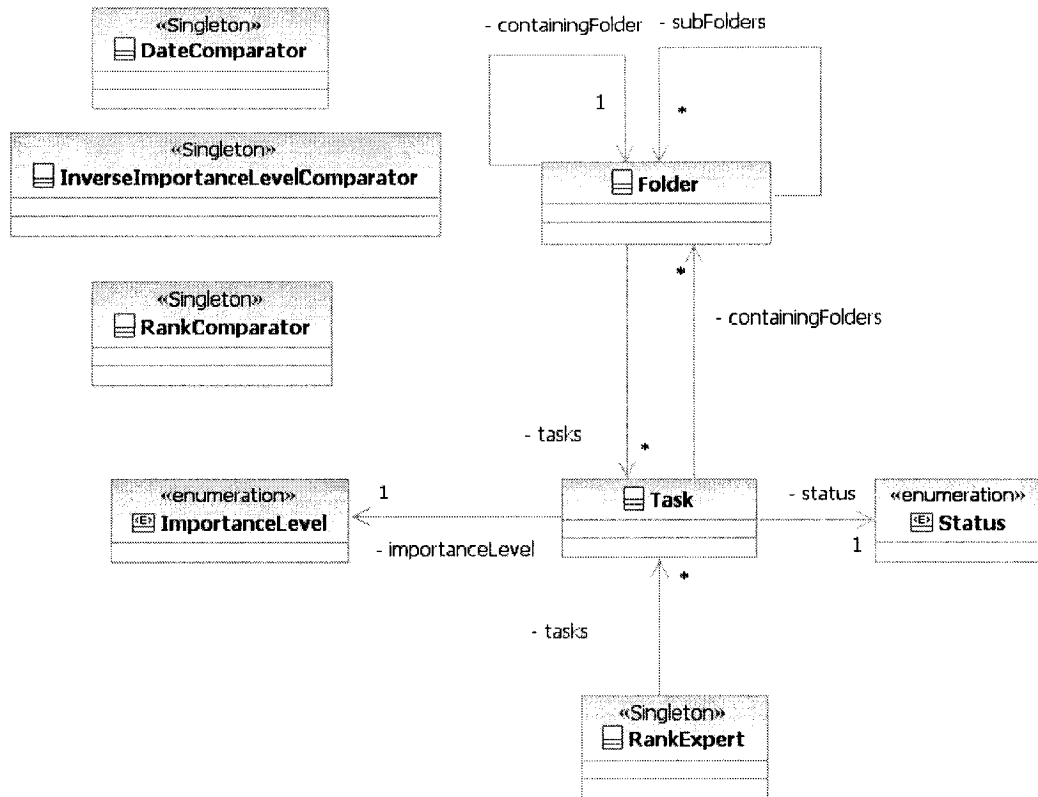


Figure 51: Class Diagram of Domain Layer

4.9 Iteration 7: Persistence

4.9.1 Introduction

In the previous iterations, we developed the domain and presentation layer for the Project Management System. In this iteration, we create the database schema for the domain entities and develop the classes required to persist the domain entities into the database.

4.9.2 Database Table Schema

We choose to use MySQL [MySQL 2007] as the database for this application. In the domain layer, `Folder` and `Task` are the entities to be persisted. We create tables `FOLDER` and `TASK` each having an `id` column that serves as the primary key. `Task` and `Folder` have a many-to-many association. This association is captured in an intermediate table `FOLDER_TASK`. This table has the fields `FOLDER_ID` and `TASK_ID` that are foreign keys to the tables `FOLDER` and `TASK` respectively. Since a folder can have at most one

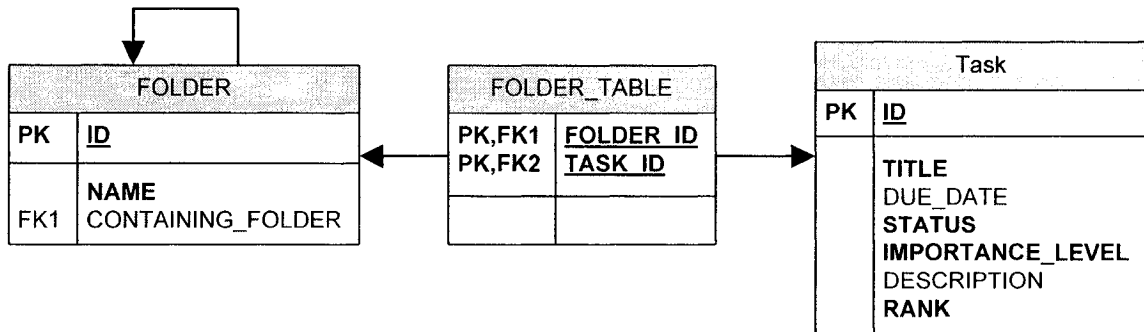


Figure 52: Database Table Schema for Folder and Task Entities



Figure 53: Class Diagram for ID generation

containing folder, we can represent this information with an additional column `CONTAINING_FOLDER` for each folder record. The columns that are highlighted in bold font (Figure 52) are columns that are required to have a value.

4.9.3 ID Allocation to Domain Entities

When a new domain entity (i.e., `Folder` or `Task`) is created, it should be assigned an id. We could create an `id` field in both `Folder` and `Task`. Alternatively, we could create an abstract super class `DomainEntity` that holds the `id` field. Creating a separate `DomainEntity` class for just containing the `id` field would be an overhead. However, we know that the `DomainEntity` is going to have added functionality when we create the `UnitOfWork` as will be discussed in Section 4.9.4. Therefore, we choose to create the `DomainEntity` class. To generate the id, we create a generic `IdGenerator` with a method `getMaxId(tableName:String):Long` that will take the domain entity name as input and get the maximum id value from the database (see - Figure 53).

4.9.4 Unit of Work

Every time a user makes a change, we could connect to the database to persist the change. This would mean connecting to the database for minor changes e.g., changing the title of a task. Such frequent connections to

the database could decrease the performance of the application. Therefore, we consider the possibility of performing a batch update on the database. When the user makes a set of changes and decides to save them, we consider that a user session has concluded and the changes are collectively persisted to the database.

In order to keep track of all the changes made, we can use the *Unit of Work* pattern [Fowler 2002]. Whenever a domain object is created, modified or removed, we can register the changes with the `UnitOfWork` using the operations `registerNew(domainObject:DomainEntity)`, `registerDirty(domainObject:DomainEntity)` or `registerRemoved(domainObject:DomainEntity)` respectively (see Figure 54). Since this functionality is common to every domain entity, we assign the responsibility of notifying the `UnitOfWork` to `DomainEntity`. The `UnitOfWork` is notified using the operations `markNew()`, `markDirty()` and `markRemoved()` which is invoked when the domain model is changed. For example, if the task title is changed using the operation `Task.setTitle(title:String)` then `setTitle(title:String)` will invoke the domain super class operation `DomainEntity.markDirty()`. Thus, the task is marked for an update to the database.

Changes like renaming the title are simple changes made to the task. However, when a folder is removed, we need to have an additional mechanism of notifying the `UnitOfWork` to remove the folder's subfolders and tasks too. Therefore, when we delete an instance `f` of `Folder`, we invoke a method `f.removeAll()` which marks all subfolders and tasks recursively for removal from the database as well. In addition, a task is marked for removal only if it has one containing folder. This is because if the task is shared by multiple folders, removing a task from one folder should not remove the task from the database.

In this manner, all the changes for the session are tracked in the `UnitOfWork`. When the user wants to save the changes, the operation `UnitOfWork.commit()` is invoked. This method is assigned the responsibility of persisting changes by calling the Data Mappers (c.f. Section 4.9.6) for `Folder` and `Task`. The Mappers update the changes to the database. Once all the changes are persisted, we clear the `UnitOfWork` of the changes with the operation `clear()`. Any change after this point, is treated as the beginning of a new session.

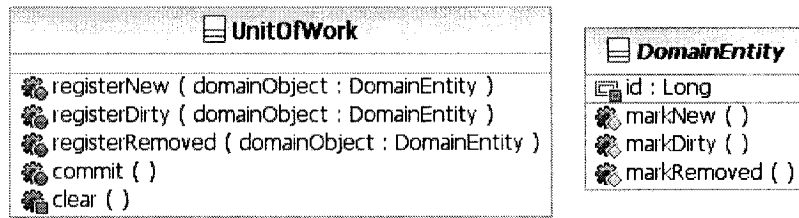


Figure 54: Class Diagram showing functionality for Batch Updates

4.9.5 Identity Map

While loading folders and tasks from the database, we face the issue of multiple instances existing for the same entity in the database. For example, if two folders *f1* and *f2* share a task *t*. When the tasks for *f1* are loaded from the database, an instance is created for *t*. However, when tasks for *f2* are loaded from the database, a separate instance will be created the same task *t*. Each instance could be updated with different values, which is not desirable. In order to prevent multiple instances of the same entity, we use the *Identity Map* pattern [Fowler 2002]. This pattern has the added advantage of caching so that we do not spend additional time asking to the database for the same entity if its instance is already loaded.

The Identity Map works similarly for `Folder` and `Task`. Taking the example of `Task`, when a `Data Mapper` (cf. Section 4.9.6) is invoked for finding a task, it will check the identity map for an instance representing the task. If the Identity Map does not contain the instance, the Mapper loads the entity information from the database, creates an instance, and then places the instance in the identity map. The behavior of obtaining the instance from the Identity Map and placing the instance in the Identity Map is performed by `getTask(taskId:Long)` and `addTask(task:Task)` respectively (Figure 55). Also, since an Identity Map should be notified if an instance of an entity is created or deleted, the `UnitOfWork` will notify the Identity Map whenever operations `registerNew()` and `registerRemoved()` of `UnitOfWork` are invoked. Since an Identity Map is relevant for a particular session, it is recommended [Fowler 2002] to place the Identity Map in the `UnitOfWork` object since the `UnitOfWork` is specific to a session. Figure 56 shows a simplified interaction between the Data Mapper and the Identity Map.

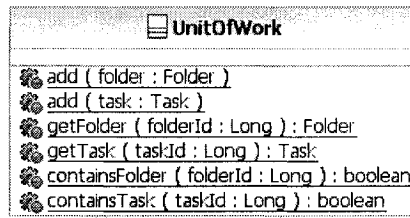


Figure 55: Class Diagram showing functionality for maintaining unique domain entity instances

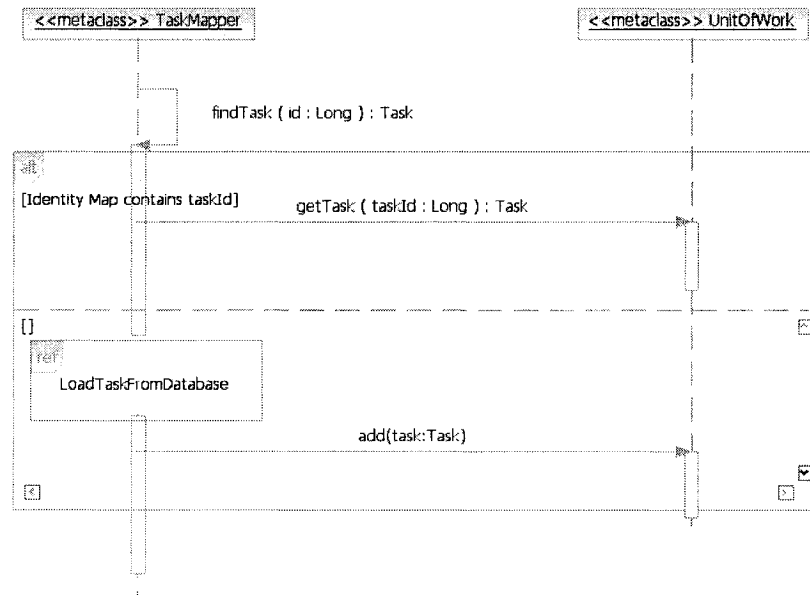


Figure 56: Sequence Diagram showing behaviour of finding a task in TaskMapper

4.9.6 Data Mapper and Table Data Gateway

A Data Mapper [Fowler 2002] is used to map data between objects and the relational database. When loading the data from the database, we first load the root folders with `FolderMapper.findRoots():Folder[*]`. We then need to determine how to load the subsequent folders and tasks. For loading tasks, we could have an operation e.g. `TaskMapper.find(taskId:Long):Task`. This operation would be called by the `FolderMapper` when it is loading the task's containing folder. However, this means that the responsibility to find the task ids for a particular containing folder will be executed in the `FolderMapper`. By the GRASP pattern *Information Expert* [Larman 2004], this logic should reside in the `TaskMapper`. Additionally, we would be creating connections to the database for each task when they can be retrieved all together with the containing folder id. Hence, we create an operation `findTasks(containingFolderId:Long):Task[*]` in the `TaskMapper`. Since the operation `Task-`

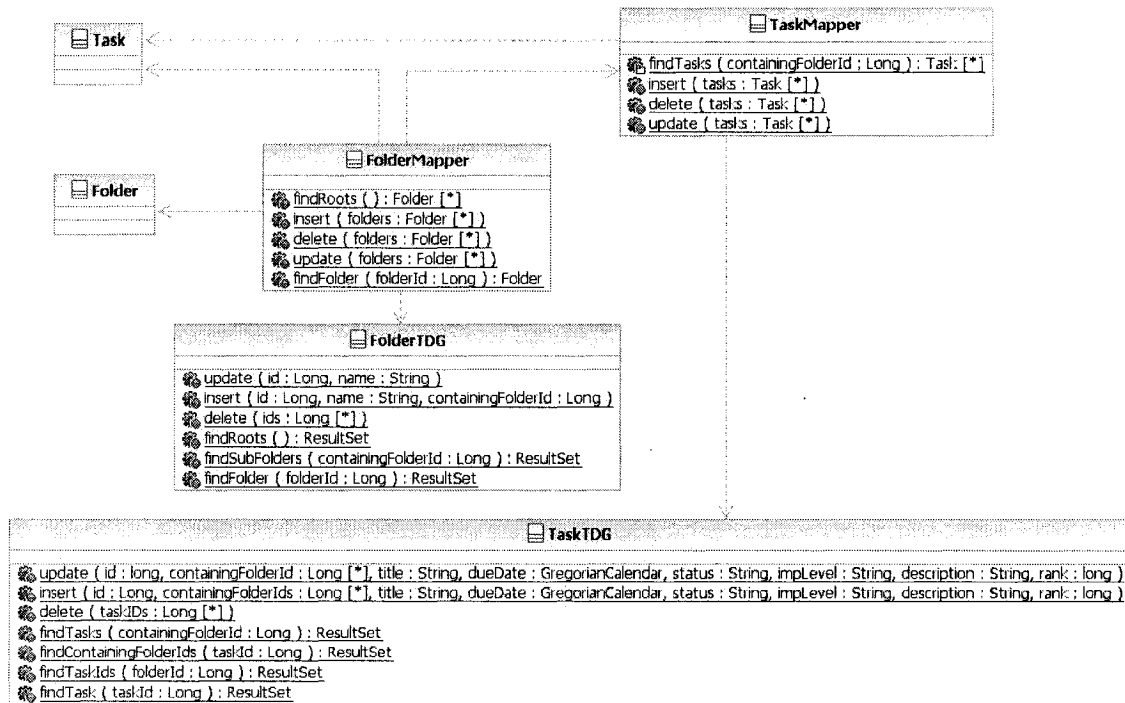


Figure 57: Class Diagram showing functionality for mapping objects to persistence

`Mapper.findTasks(containingFolderId:Long):Task[*]` is invoked by the `FolderMapper`, it is package visible. We similarly create an operation `findSubFolders(containingFolderId:Long):Folder[*]` as well (See- Figure 57). In addition, we also create an operation `findFolder(folderId: Long):Folder` in `FolderMapper` which will be required for lazy loading as will be explained in Section 4.9.7. We could place the SQL queries to be executed in the appropriate Mappers. However, for a cleaner separation of responsibility, all SQL queries are placed in a separate class that applies the Table Data Gateway (TDG) Pattern [Fowler 2002] (See- Figure 57). We also decide about the important responsibility of who will maintain the mapping between a folder and its tasks. Either the `FolderTDG` or the `TaskTDG` can maintain the mapping. We choose the `TaskTDG` to manage the mapping that allows us to create the folder records in the `FOLDER` table first.

4.9.7 Virtual Proxy

The classes `Folder` and `Task` have a bidirectional association between them. This can lead to an infinite loop when we load the database records into memory. For example, consider a folder F that has a task T . If the folder F is to be loaded into memory, its fields are to be populated. Since T is a task of F , an attempt is

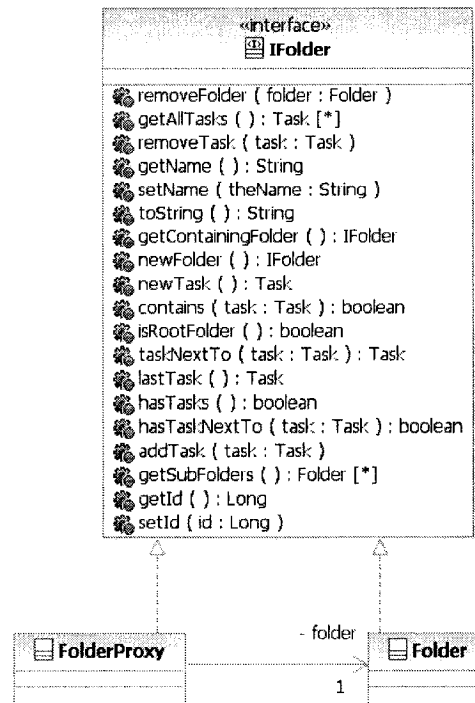


Figure 58: Class Diagram showing functionality to prevent deadlock for loading domain entities

made to load *T*. However, *T* has a containing folder field to be loaded where it attempts to load *F*, which leads to an infinite loop. A similar problem is faced for a folder and its subfolders as well.

In order to prevent this, we use the *Proxy Pattern* [Gamma et al. 1995], which allows us to lazily load one of the roles in the bidirectional association. We could create a proxy class for both `Folder` and `Task`. Therefore, when we load a folder, we can create proxies for its subfolders and tasks. The other option is to create proxies only for the containing folder. Since the second option restricts the creation of a proxy to a single domain entity, namely `Folder`, we create a `FolderProxy`. Figure 59 illustrates how a proxy is created by a `Mapper`. When any operation of the `FolderProxy` instance is invoked for the first time, the `FolderProxy` instance loads the folder using `FolderMapper.findFolder(folderId:Long):Folder` and delegates all calls to the folder instance.

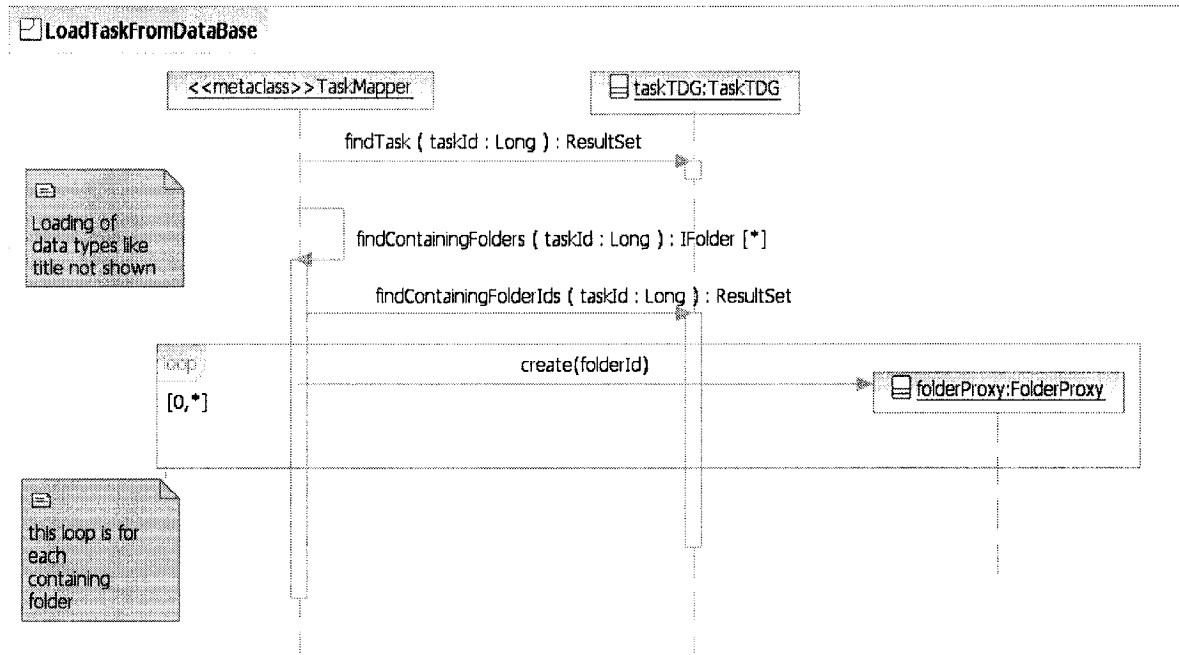


Figure 59: Sequence Diagram showing behavior of loading a task from the database

4.10 Case Study Summary

In the previous section, we completed the final iteration in the case study. In this section, we summarize the work done in the previous iterations and provide a high-level view of the complete model.

In iteration 0, we defined the initial domain model for the Project Management System (PMS). We identified the domain concepts such as, `Task`, `Status`, `ImportanceLevel`. We brainstormed to find a concept that could be used to group tasks and settled on the concept of `Folder`. We decided that a task can only belong to one folder and that tasks within a folder should be ordered.

In iteration 1, we selected the basic features to model. They involved creating, deleting and editing of tasks and folders. To satisfy the constraint of ordering tasks, we decided to allot each task a rank. We demonstrated how to use the RSA support for design patterns by generating a `RankComparator Singleton`. We also demonstrated how RSA generates code from the model.

In iteration 2, we implemented the basic UI for the features implemented in iteration 1. We introduced the Eclipse Rich Client Platform (RCP) as our UI toolkit. In this iteration, we chose not to populate the domain objects with event notification code. Hence, we created wrappers around the domain entities to take care of

the notification responsibilities. It was in this iteration that we settled on a table view to show tasks and a tree view to show the folder hierarchy.

From iteration 3 onwards, whenever we implemented a feature, we developed both the domain layer and the presentation layer in the same iteration. In this iteration, we provided the feature of viewing all the tasks in a folder including the tasks in folder's subfolders as well.

Once we became familiar with the application of MDD, we chose to introduce TDD into the project. We applied TDD for iteration 4 where we developed the functionality of sorting tasks by their importance level and due date. The code was written incrementally to satisfy each failing test. We first developed the tests, code for the different comparators, and then did the same for the sort functionality. We then created the UI functionality for displaying tasks in a sorted order. We discovered that TDD and MDD both incrementally evolve the code but work at different levels of abstraction. The tests work at the level of code while the model is at a higher level of abstraction. Since our focus is on MDD, we returned to MDD in subsequent iterations.

In iteration 5, we developed the functionality of reordering tasks. We knew that we wanted to use the Eclipse JFace Drag and Drop (DnD) feature but were not familiar with the API. Hence, in contrast to previous iterations, we performed an initial UI investigation before designing for the feature. We created a `RankExpert` that would reorder tasks.

In iteration 6, we decided to develop the functionality of sharing a task across multiple folders. We relaxed our initial constraint that a task can only belong to one folder. Since we were already familiar with the Eclipse JFace functionality of Drag and Drop, we utilized it for sharing tasks between folders as well. Noticing a major duplication of the UI context menu for both the `FolderView` and the `TaskView`, we refactored the code into a separate `ContextMenu` class.

Once the domain layer stabilized, we developed the functionality to persist the folder and task entities into a database, taking inspiration from enterprise application architecture patterns [Fowler 2002]. First, a database schema was created for the `Folder` and `Task` entities. This influenced the design of the `UnitOfWork`, the Data Mappers, and the Table Data Gateways. We created a `UnitOfWork` to perform transactional updates to the database. Faced with the challenge of maintaining a single instance for each

task entity, we introduced an Identity Map. Due to the bidirectional association between `Folder` and `Task`, we created a Virtual Proxy [Gamma et al. 1995] for `Folder` so that we could prevent infinite loops while loading folders and tasks. The entire complexity of mapping objects into the database was encapsulated in the Data Mappers and Table Data Gateways. The Data Mappers hid the complexity of interacting with the Identity Map, creating `FolderProxy` instances, and interacting with the Table Data Gateways.

The following diagrams give a high-level view of all the classes used in this case study. First, Figure 60 shows the three-layer architecture used in this case study i.e., the presentation layer (labeled UI), the domain layer, and the technical services layer. Figure 61 shows the presentation layer details and Figure 62 show the domain and technical services layer details. Figure 63 shows the tests written for the domain layer.

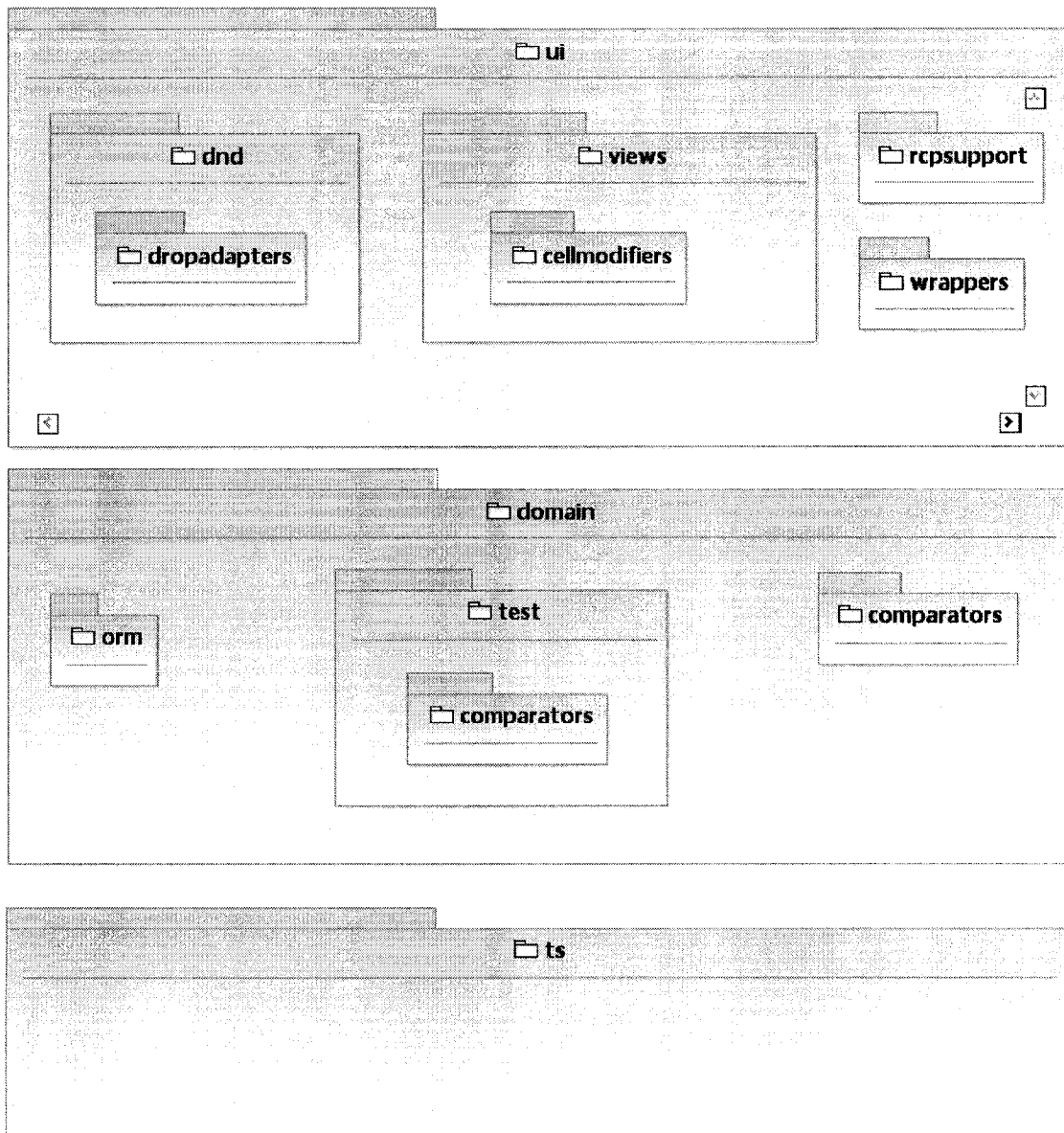


Figure 60: Package Diagram for the Project Management System

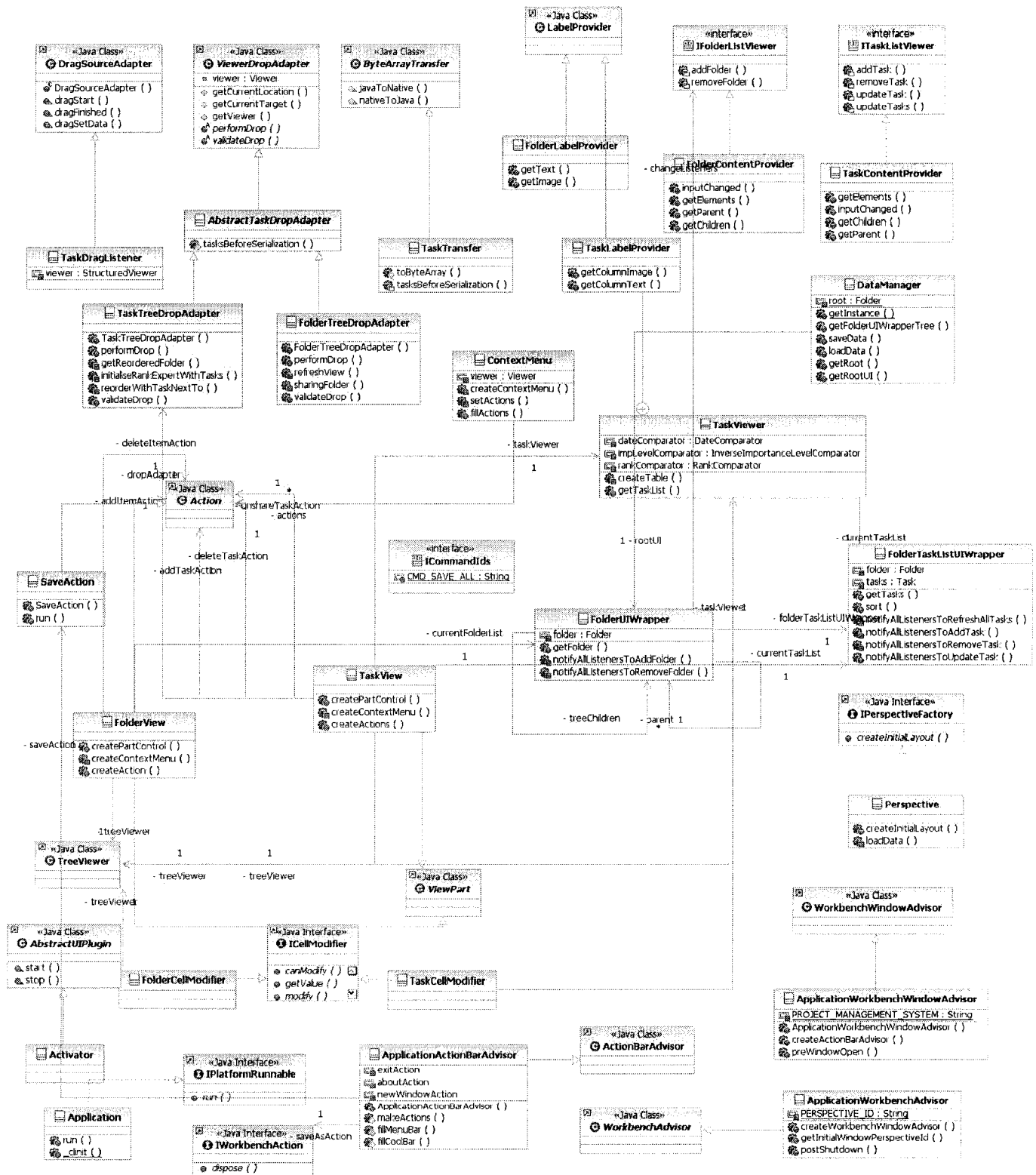


Figure 61: Class Diagram of the Presentation Layer

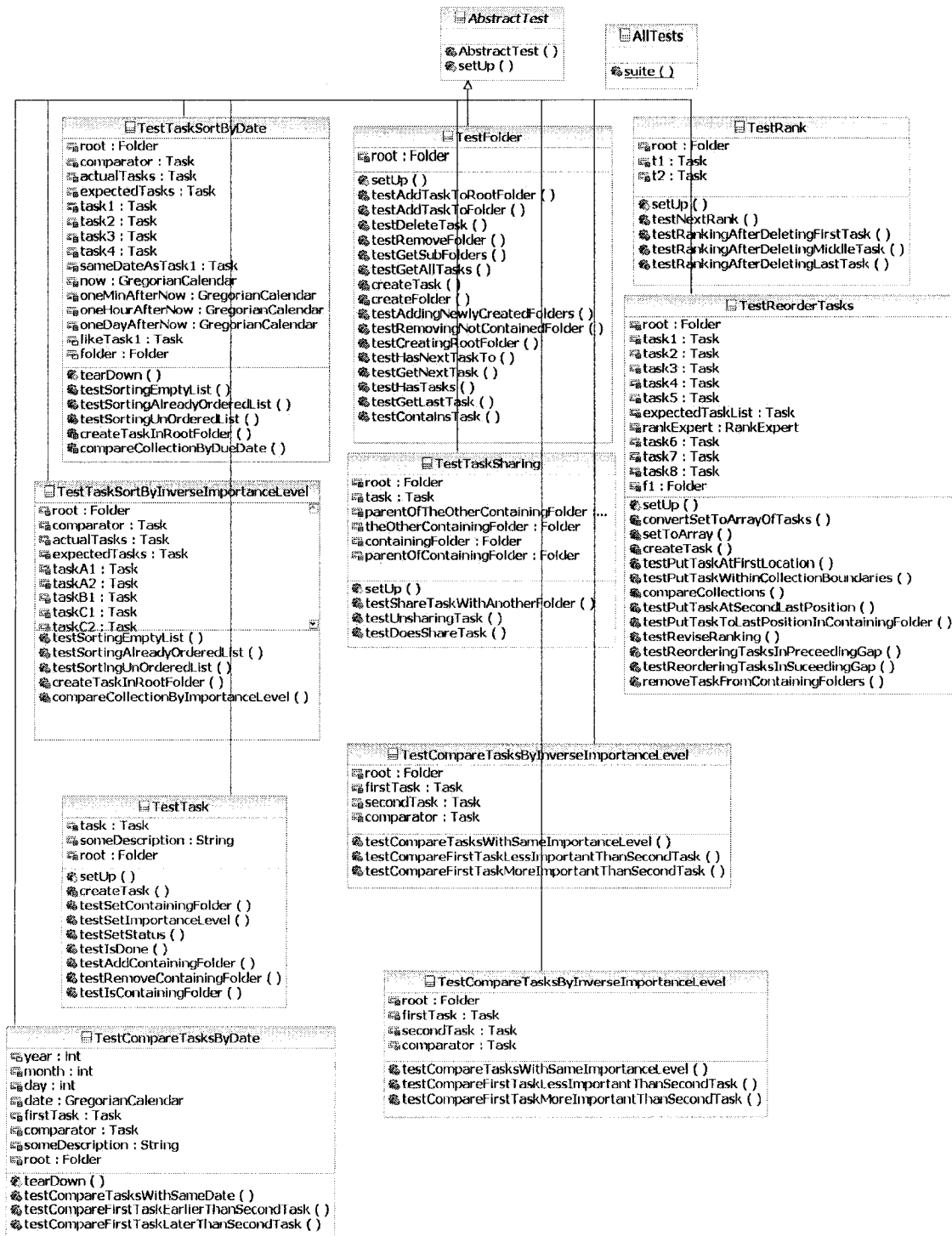


Figure 63: Class Diagram showing Unit Tests for the Domain Layer

During the application of MDD in this case study, certain limitations of RSA were encountered. Their impact on MDD and guidelines to work around these limitations are suggested in the next Chapter.

5 RSA Limitations and Guidelines

In this chapter, the limitations of RSA that were discovered during the case study are highlighted. It is important that these limitations are resolved if the practice of MDD is to be integrated into mainstream development. This chapter is useful to developers interested in adopting MDD because some of the limitations described in this chapter will influence the mode in which RSA is utilized for MDD. Some of these limitations restrict the use of the tool while others restrict the use of Java language constructs. If guidelines for a limitation are possible, they are suggested after the limitation. The chapter then discusses general guidelines for the application of MDD with RSA. The guidelines will aid the developer in using RSA for MDD.

5.1 Reverse Transformation Done For Entire Java Project

For a reverse transformation from code to model, the source is an entire Java project. This means that even if we make a small change to code, we have to run the reverse transformation for the entire Java code, even for those modules that have not been changed at all. Figure 64 shows an example of the conflict merge window shown by RSA during a reverse transformation. If the developer merging the models is very familiar with the code and model, it is possible to merge the models. However if the developer is not very familiar, merging models may be an effortful task due to the many changes that are reported by the reverse transformation.

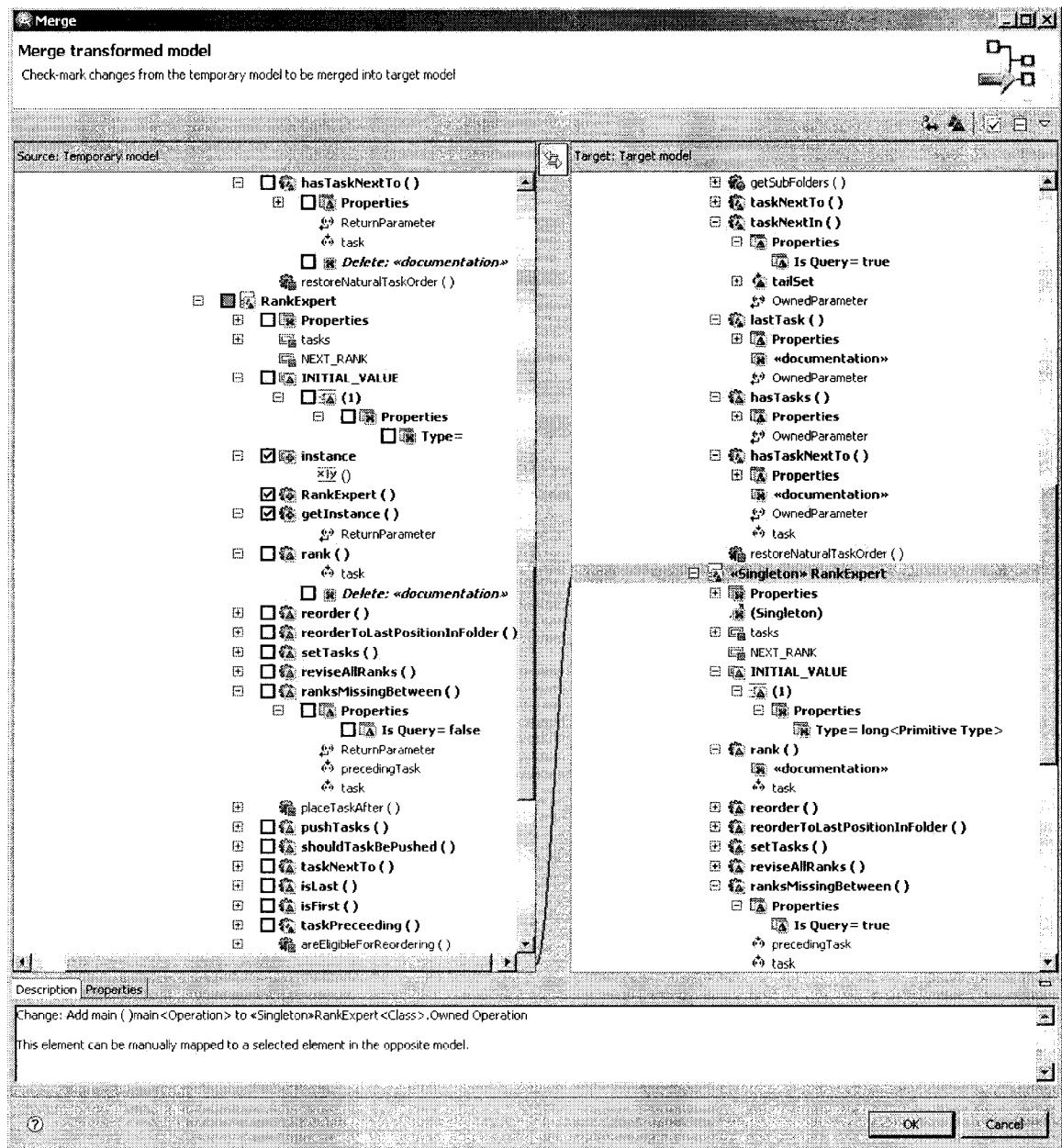


Figure 64: Conflict Merge during Reverse Transformation from code to model

Therefore, modifications to the code should be done in very small increments before reverse transforming the code. This will reduce the complexity of merging during reverse transformation. Fortunately, for forward transformation from UML model to Java code, the source for the transformation can be as small as a UML class. This feature should be used to make changes to a class or a small set of classes and immediately forward transform those classes into code.

5.2 Synchronization Issues in Reverse Transformation

5.2.1 Limitation: No Link Created by Reverse Transformation for Added Code.

RSA allows a developer to have class members (i.e., class fields and methods) that are added directly to the code to be incorporated in the UML model using the Java-to-UML reverse transformation. After reverse transformation to the UML model, there is no distinction between members added directly to the code and those added directly to the model.

A problem arises when a member is deleted from the model and the developer runs the UML-to-Java transformation. If the deleted member was initially created in the model then it is deleted from the code. However, if the deleted member was initially created in the code (and hence does not possess an `@generated` tag), it would not be deleted in the code, contrary to the expectations of the developer who has removed it from the model. The developer has to remember to remove the method from the code as well. If the method is not deleted from the code then the model and code are not synchronized. Worse, if the developer has created an alternative solution for the same functionality then there may be two solutions to the same problem, which is a form of duplication.

5.2.2 Importance of Reverse Transformation for the Adoption of MDD

If we assume that MDD is still in the early stages of adoption by industry then it would be safe to say that significant code generation and model reuse is limited. Hence, unlike mature compilers which abstract assembly language to the extent that developers no longer need to work with assembly language, the same cannot be said of tools that manipulate models. Therefore, it would be appropriate to note that a significant if not major part of the software development done would still be at the implementation level. This includes development teams and projects whose main interest in MDD is the communication of design intent and high-level documentation to maintenance developers, which in their own right is a valuable benefit for large-scale projects. In such cases, it is highly possible that along with code modification, current software practices like refactoring and agile methodologies like TDD are practiced where most of the work is done directly with the code.

Additionally, not all design may be done up front. Certain design abstractions may emerge from code refactorings. For example, if we create methods in different classes which perform the same operation, such

duplication would be visible at the model level. This improvement in design occurs because enough refactoring happened at the code level. Therefore, it becomes imperative for MDD tools to take into consideration working directly with code until code can be completely generated from models.

5.2.3 Workarounds for Reverse Transformations in RSA

Since reverse transformation from code to model in RSA is still not mature, certain guidelines have to be devised so that developers can synchronize modified code and the model. However, before we suggest ways to work around the limitation of reverse transformation, it must be understood that this is not the way the RSA's documentation encourages developers to maintain model-code synchronization. According to the documentation, all structural changes should be made to the UML model instead of to the generated code.

If teams are interested in working at the code level but still maintaining model-code synchronization, then the following basic strategy should be considered if additions are made to the code:

- Every time a field or method is created in code, we also create the tags `@generated`, `// begin-user-code` and `// end-user-code`. The `@generated` tag is required to establish the link between the model and code. The `// begin-user-code` and `// end-user-code` are required to prevent the code in the method body from being deleted once the link is established.
- After creating the link for the new fields and methods in the code, reverse transform the Java project back into the associated UML model. This reverse transformation should be done before any future forward transformations. If the reverse transformation is not performed, the subsequent forward transformations will remove any changes made in the code.

The following subsections suggest some more ways for working directly with code and maintaining model-code synchronization.

5.2.3.1 Create Field

Creating a field in the code is discouraged, as the effort to create a field in the UML model is minimal. However, if the field was created as a part of a refactoring, then the `@generated` tag can be generated by

Eclipse using the key combination of Alt+Shift+J (see Appendix). This change should then be reverse transformed back into the model.

5.2.3.2 Create Method

For adding methods directly in code, there are two common ways to do so other than manually typing a new method signature and body:

- While writing code, the developer feels the need for another method and types a call to it. The developer then uses the Eclipse Quick Fix feature (as explained in 4.6.2) to create the method with an empty body.
- The developer can select a section of code and create a new method using the “Extract Method” option in the Refactor menu.

For both of these techniques, we can automate most—if not of all—of the work of creating the tags for every method created in the code by customizing Eclipse *Preferences* (see the Appendix). By customizing the Eclipse *Preferences*, every time a developer uses the Eclipse Quick Fix feature to create methods, Eclipse will automatically generate both the @generated tag in the Javadoc comments and the `// begin-user-code` and `// end-user-code` tags within the method body.

However, while extracting a method using the Refactoring Menu, the developer has to follow the advised workflow so that the refactored code is not deleted. Once the method is generated, the method body can be entered and then the Eclipse Content Assist feature invoked by pressing `Ctrl + <Space>`. Eclipse will show a Content Assist Dialog (see Figure 65). Type “tag” and press *Enter*. The `// begin-user-code` and `// end-user-code` tags will be generated in the editor. The method body should be then copied and pasted within these comments. Note that failure to do so will erase all the code within the method the next time the model is forward transformed. Hence, though such a template eases the work of maintaining refactoring code, this technique should be used with caution.

```

private static Task findTask(Long id) throws SQLException {
    Task task1;
    if(UnitOfWork.containsTask(id)){
        task1 = UnitOfWork.getTask(id);
    }
    else{
        task1= loadTaskFromDatabase(id);
        UnitOfWork.add(task1);
    }
    return task1;
}

```

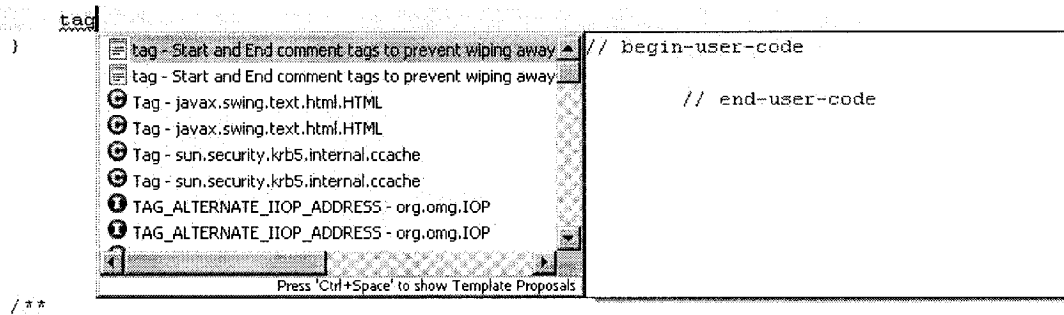


Figure 65: Eclipse Content Assist for Tag Template

It is important to perform the next step only after placing the method body between the `// begin-user-code` and `// end-user-code` comments comment template: place the cursor within the method signature and generate the `@generated` tag by using the key combination of `Alt+Shift+J` (see the Appendix). If this step is done without placing the method body within the comment template, then the subsequent forward transformation will wipe away the method body.

After making a series of refactorings, always perform a reverse transformation to update the UML model with the changes in the code. If this is not done, then the subsequent forward transformation will delete the changes made in the implementation.

5.2.3.3 Rename Class Members

Renaming Class properties and operations in the UML model does not refactor the fields and methods in the corresponding Java class as expected. All references to the previously renamed field or method will still exist in the code. Eclipse marks these references as compilation errors.

Therefore, we have to use the refactoring capability of Eclipse to make the required renaming in the code itself. For propagating the change back to the model, it is simpler to change the name directly in the UML model rather than to reverse transform the modification back to the model.

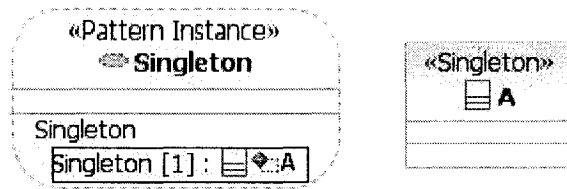


Figure 66: Singleton Profile in RSA

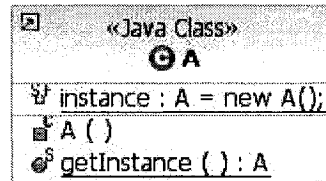


Figure 67: Visualization of generated Java Class A

5.3 Reverse Transformation of UML Profiles

In our case study, we had the opportunity to use the *Singleton* pattern [Gamma et al. 1995]. Figure 66 shows a class A after the application of the Singleton profile. After forward transformation to code, the appropriate class fields and methods were generated (RSA generated visualization shown in Figure 67).

However, if we reverse transform the same code without making any changes at all, the class properties and methods generated in the code by the design pattern profile are not recognized as a part of the same design pattern profile but are instead treated as independent properties and methods. Figure 68 shows the merging window during this reverse transformation. The temporary model generated from the source code is on the left and the existing UML model with which the changes are to be merged is on the right. Since the attributes in the code were generated from the profile itself, there should be no conflict. However, the merging window shows a conflict and suggests deleting the UML Singleton Profile (bottom left of Figure 68). If we retain the profile and deselect the fields and methods in the temporary model, RSA will report the same conflicts every time the developer performs a reverse transformation. If we choose to delete the profile information then the class loses all profile information except the *Stereotype* declaration.

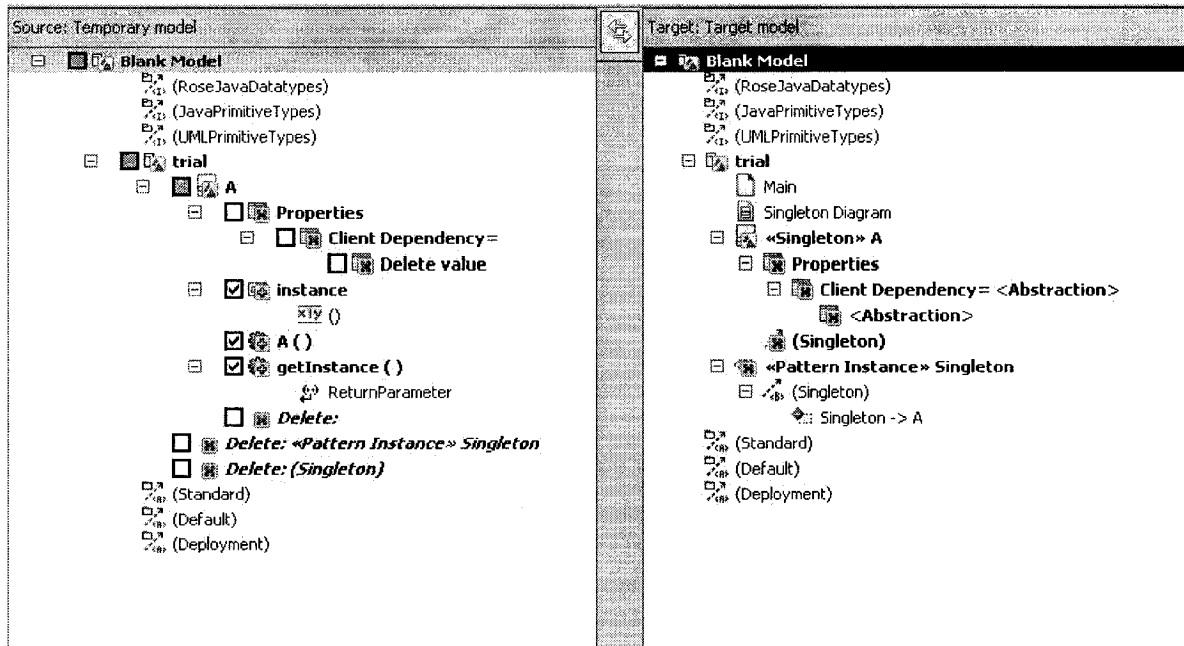


Figure 68: Model Merging Window during Reverse Transformation from Java to UML for the Singleton Pattern

Maintaining profile information in models is important because it can provide the following benefits:

- To reuse domain knowledge captured as abstract concepts in the model.
- To generate significant portions of code using models.
- To reuse best practice design solutions, like Design Patterns [Gamma et al. 1995].

Profiles may be used in large-scale industry projects where teams have expertise in building a family of similar applications. Such teams may choose to capture their domain knowledge and design choices using profiles. These profiles would then be used to generate a significant number of classes in code. Due to the large number of classes created, reverse transformations could be a source of excessive overhead with respect to model-code synchronization (as seen in Figure 64). Thus, if RSA is to be used effectively by teams with mature domain or design models for the purpose of code generation, they will have to refrain from modifying the structure of classes directly in the code. This restriction would impede practices like refactoring and TDD where developers interact directly with the code.

5.4 UML-to-Java Transformation Impacts

After an initial model is transformed into code and the required methods are filled, a developer can make changes to **code** directly. Once the code is modified, if the same model is again transformed to code, certain modifications in the code are impacted. A detailed table of the impact on code is mentioned under the heading “*Impact of code changes on UML-to-Java transformation output*” in RSA documentation [RSA Help 2007a]. An abridged version of the RSA documentation table, enhanced with comments inspired by the author’s experiences, is presented in Table 1. The comments elaborate on the full impact on code.

Changed code element	Change	Impact of rerunning the transformation	Additional Comments by Thesis Author
Class or interface	Add implementation or extension	Added implementation or extension is removed.	
Class or interface	Remove implementation or extension	Previously generated implementation or extension is restored.	
Field	Move	Field is removed from the new location. Field is restored to the previous location.	References to the Field in the new location are still present and show as compilation errors.
Field	Rename	Previously generated field is restored. Renamed field is removed	References to the Field in the class are still present and show as compilation errors.
Method	Modify return type	Previously generated return type is restored.	
Method	Modify signature	Previously generated method and signature are restored. Method with the new signature is removed	The method body in the method with the new signature is removed as well.
Method	Move	Method is restored to the previous location. Method in the new location is removed.	All method calls to the method in the new location are still present and show as compilation errors
Method	Rename	Previously generated method is restored. Renamed method is removed	The method body in the renamed method is removed as well.
Qualifier	Modified	Previously generated qualifier is restored.	

Table 1: Impact of UML-to-Java transformation on modified code

Similarly, after an initial model is transformed into code and the required methods are filled out, a developer can make changes to the **model**. If the code is modified and the changed model is again transformed to code, the code can be affected [RSA Help 2007b]. Details are given in Table 2.

Model element	Change	Impact on previously generated code when you rerun the transformation	Comments by Thesis Author
Class or interface	Add	File overwrite option of the UML-to-Java transformation determines whether the previously generated Java file is deleted. New Java file is created	
Class or interface	Add implementation or extension	Clauses are rewritten or added.	
Class or interface	Remove implementation or extension	Clauses are rewritten or removed.	
Field	Move	Field in the previous location is removed. Field is added in the new location	References to the Field in the previous location are still present and show as compilation errors.
Field	Rename	Field with the previous name is removed. Field with the new name is added	References to the Field with the previous name in the class are still present and show as compilation errors.
Method	Modify return type	Method return type is updated	
Method	Modify signature	Method with the previous signature is removed. Method with the new signature is added	Even the method body in the method with the previous signature is removed
Method	Move	Method in the original location is removed. Method is added in the new location	All method calls to the method in the original location are still present and show as compilation errors
Method	Rename	Renamed method is added	The method with the previous name and its method body is removed

Table 2: Impact of UML-to-Java transformation on code if model is changed

5.5 Sequence Diagram Support

Dynamic modeling has been used frequently during this case study. For example, sequence diagrams were very useful for modeling the interaction with the `FolderMapper` for finding a folder. This involved reasoning about creating proxy folders for lazy loading, the interaction with the `UnitOfWork` for accessing the Identity Map, interacting with the `TaskMapper` for loading tasks, and more importantly, reasoning about recursion since folder have subfolders. It would have been very impractical to reason about all these interactions using class diagrams alone. Therefore, in MDD, dynamic modeling should be frequently used to reason about complex behavior.

However, the sequence diagrams are not created using RSA's sequence diagramming support. RSA is impractical for quick, initial dynamic modeling. This is because using the tool breaks the flow of thought

and is not as intuitive as class diagram modeling in RSA. Secondly, we often make changes to the sequence diagram as the interactions become clearer. It is cumbersome to make changes to a sequence diagram created within the tool. Thus, it is advisable to create sequence diagrams on paper and manually map each message call in the sequence diagram to a method in a UML class diagram within the tool.

5.6 Java Specific Limitations

5.6.1 Accessor and Mutators

In RSA, one can create getters and setters for class properties by selecting an option in the UML-to-Java transformation configuration file. However, there may be certain private fields in classes for which the developer may not want getters and setters to be generated. Presently, there is no way to indicate that getters and setter should not be generated for certain class fields. The developer *can* deselect the option to create getters and setters for class properties in the configuration file, requiring getters and setters to be created directly in the code by hand. The problem with the approach of creating methods in code has already been mentioned above.

If we choose not to use the RSA transformation option of generating getters and setters, then we can use the basic Eclipse feature to generate getters and setters (available from the Source Menu). The `@generated` tag in the Javadoc comment and method body tags (`// begin-user-code` and `// end-user-code`) can be created automatically by customizing Eclipse Preferences (see the Appendix). The Java project should then be reverse transformed so that the getters and setters are introduced in the corresponding UML model.

5.6.2 Lack of Complete Support for Java Generics

In the case study, parameterized comparators were created with the help of Java Generics. The class type information for `RankComparator` is provided in Figure 69.

```
public class RankComparator implements Comparator<Task>,Serializable { ...
    public int compare(Task o1, Task o2) {
        ....
    }
}
```

Figure 69: Class Signature of RankComparator

```

public void sort(Comparator comparator) {
    Comparator<Task> com = (Comparator<Task>) comparator;
}
*/

```

Figure 70: Illustration showing a workaround for generics in method parameters

`RankComparator` implements the `Comparator` parameterized with `Task`. This is necessary in order to realize the type `Task` in the method interface `compare(o1:Task, o2:Task)`. When we reverse transform from Java to UML, the information that `RankComparator` implements a `Comparator` is not present in the UML model. However, if `RankComparator` had implemented a non-parameterized comparator (e.g., `public class RankComparator implements Comparator {...}`), then the information would have been present in the UML model after reverse transformation.

This limitation extends to method parameters and class properties as well. For example, if a method `sort(comparator:Comparator<Task>)` is added to the code then after reverse transformation the method signature in the UML model would be `sort(comparator:Task)`. The type information changes during reverse transformation. Hence, semantic information is lost and the model have inconsistent information with respect to the code.

Therefore, RSA only supports parameterization of collections of objects i.e., the fields and method parameters that are to hold multiple values. Non-collection Java types (e.g. `Comparator<T>`, `HashMap<K,V>`) cannot be parameterized. If model-code synchronization is required then non-parameterized types have to be used. If parameterization is important, one can remove the `@generated` tag above the corresponding class or field. This will unlink the field or class from the model, thus allowing the use of parameterized types. For method parameters, a parameterized argument can be passed to a non-parameterized formal parameter, which is type cast to the required parameterized type (see Figure 70).

5.6.3 Method Parameters and Return Values

In RSA, a method parameter having multiplicity greater than one can be qualified using the *isOrdered* and *isUnique* options in the UML model. Based on the selection of these options, the appropriate Java collection is created during forward transformation from UML to Java.

The limitation pertains to using collections for method parameters in the UML model. If the parameter is a collection and is the last (or only) formal parameter, then the method will be transformed to a variable arity method in Java (*varargs*) after forward transformation. Irrespective of the qualification of the collection, the method will only accept an array or variable number of arguments of the same type. It will not accept Java collection types.

The implication of this limitation arises when the developer creates a method directly in the code with a collection parameter as explained above and reverse transforms the Java project into a UML model. Note that such a method will not have an `@generated` tag associated with it. When the same UML model is forward transformed into the Java project, a new method with same name but with variable arity parameter is created. Thus, the class will have two methods with the same name but different parameter types. One must be removed. If not then, during another reverse transformation, there will be two methods with the same name and signature in the UML model. If we forward transform from model to code again, there will be three methods: two with variable arity and the original method with a collection as a parameter type.

In order to prevent such cases, the developer should follow the discipline of using *varargs* for the last (or single) formal parameter if a multiplicity greater than one is required. In addition, a method cannot have an array as a return value. If a method has to return multiple instances, collections have to be used. Therefore, if multiple instances are returned as a collection by a method, the collection has to be converted into an array before passing it to a variable arity method. Eclipse has a convenient *template* called `toArray` that helps to convert collections to arrays.

5.6.4 Lack of Support for Enumeration Operations

In RSA, an operation can be added to an enumeration in the UML model. However, during forward transformation, if the implementation for the enumeration is generated for the first time, only a method signature is generated for the operation but with compilation errors. For example, suppose an operation `enumMethod()` is added to an enumeration `E` in the UML model. If `E` is forward generated for the first time, then the method signature `public void enumMethod();` is generated. Since there are no curly braces for the method, the compiler detects it as an error. If the operation was added to an enumeration in the model whose implementation already exists, then on forward transformation, the operation is not

generated in the implementation at all. In addition, methods that are added to an enumeration in the code itself are not recognized by the reverse transformation. Support for displaying operations in the UML model would be useful because Java 5 now supports the `enum` construct that can have arbitrary fields and methods.

5.6.5 Final Keyword for Method Parameters

It is considered bad practice to use method parameters as working variables [McConnell 1993] i.e., method parameters should not be assigned to within the method body. In Java, this can be prevented by making the method parameter a constant by affixing the `final` qualifier just before the method parameter declaration e.g., `public myMethod(final int x) {...}`. There is support in RSA to qualify a class property as a constant using the *Leaf* option as shown in Figure 12. However, the same cannot be done for method parameters. Even if the developer affixes the `final` qualifier to the method parameter declaration in the code directly, subsequent forward transformations will remove the `final` qualifier.

5.7 General Guidelines

5.7.1 Model Public Interfaces First

While designing for a feature, model the responsibilities (public methods) for classes and important private methods first. There is more value modeling detailed private methods of the class during implementation. This is because private methods are also used for purposes other than public responsibilities of a class. Small methods are used to increase the understandability of the program. Methods can be used to abstract algorithms internal to the class. They can also be used to remove duplication, which may not be evident when designing the class structure at a higher level. The point to be stressed is that such methods emerge during implementation rather than during design.

For example, while designing for the reordering feature, we first settled on the public interface for `RankExpert` as shown in Figure 71. However while writing the algorithm for `RankExpert.reorder()`, we decided to perform the refactoring *Consolidate Conditional Expression* [Fowler 2000] and added the private method `areEligibleForReordering`. We also introduced methods like `RankExpert.placeTaskAfter()` and `pushTasks()` for better understandability as they hide the complexity of

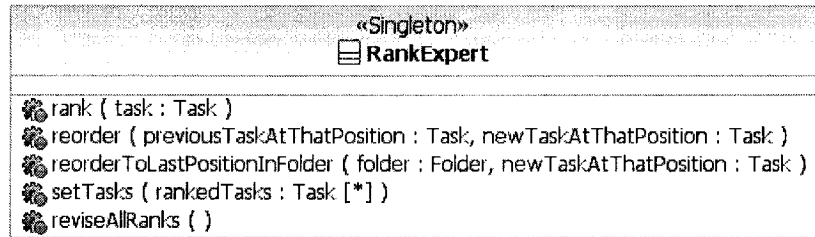


Figure 71: Public methods of RankExpert

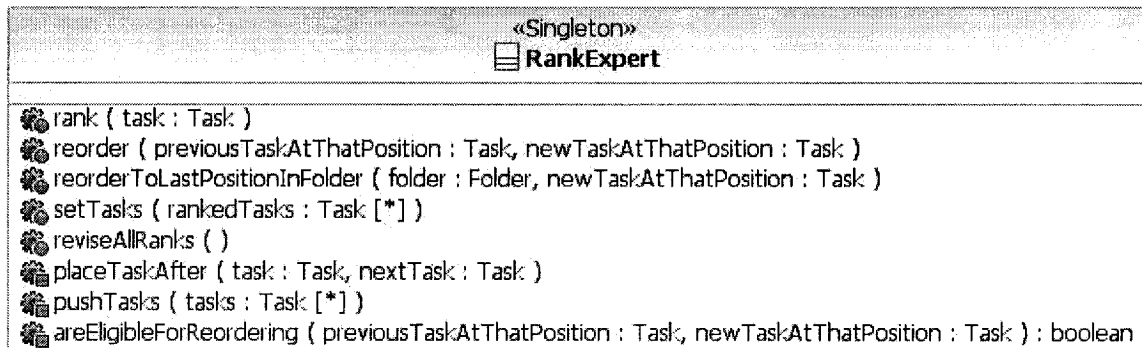


Figure 72: Evolution of RankExpert during implementation

manipulating the rank. A partial evolution of the model is shown in Figure 72. Thus, these methods originated during the implementation phase.

For teams, which are interested in code generation and usage of profiles, addition of such methods would have to be done at the model level. However, those teams more interested in using models for communication of design intent and maintenance can perform reverse transformations using the workarounds mentioned earlier.

5.7.2 Detecting Erasure of Code

Even if all the guidelines and workarounds are followed, it may very well happen that a method body is erased during a forward transformation (from model to code). Some of the scenarios where the method body can be wiped out include:

- The developer accidentally removes the `// begin-user-code` and `// end-user-code` comment tags within the method body.
- The developer performs refactoring and forgets to put the `// begin-user-code` and `// end-user-code` comment tags.

- The developer performs refactoring but forgets to reverse transform the modified code back to the model.
- The developer comments out a method but not the `@generated` tag that is associated with the method. The `@generated` tag will be associated with the next method in the source code. If the next method was added directly in the code—and hence may not have `// begin-user-code` and `// end-user-code` comments—then that next method will be erased during the next forward transformation.

Some suggestions to detect erasure of code are given below:

- If unit tests are maintained, execute them after making forward transformations of the entire model. The operations whose method bodies are erased will fail the tests.
- When RSA generates an empty method body, we can also generate code to throw an `UnsupportedOperationException` as shown in the Appendix. This exception will be thrown during the execution of the method, allowing us to detect that the method body has been erased.

5.7.3 Undo Forward Transformation Effect

In all cases where the forward transformation has caused undesirable effects, the developer can revert the source code to its state before the transformation by using the Eclipse feature *Replace with previous from local history* (one way to access this is by right clicking in the editor frame of the file to be reverted. In the context menu that shows up, select *Replace with → Previous from Local History*). In addition, if the developer has the habit of versioning source code in short increments, then the latest generated code can be replaced by the last version of the file from the version repository.

6 Conclusions and Future Work

The major contribution of this thesis is a detailed case study describing the application of MDD to the development of a non-trivial software application. We developed the application as a series of iterations thus showing that MDD can be used in an iterative process. This was demonstrated by performing MDD in very short increments of modeling, transforming model to code and then working out the implementation details.

We demonstrated how a developer can think and communicate about software development at a higher level of design using MDD. The details of round-trip engineering and implementation were hidden in the background. Thus, this text serves as an example of how developers can manage complexity by abstracting from the implementation details with the help of models.

We described the linking mechanism by which RSA attempted to solve the problem of model-code synchronization. As explained in Section 5.2, this linking mechanism did not provide complete round-trip engineering. Guidelines have been provided in Chapter 5. We also demonstrated the potential of software reuse by utilizing design pattern transformations supported by RSA. In addition, we came across relevant practical issues that RSA needs to address in future versions. These practical issues include the lack of support for Java Generics, suppression of code generation of getters and setters for class properties and the overhead of converting collections into Java arrays when passing them as method parameters. Realizing that code-centric activities like refactoring can be performed by developers during MDD, we demonstrated how to use the refactoring feature of Eclipse and still maintain model-code synchronization for simple refactorings.

We applied another code-centric activity namely, TDD to one of the iterations in our case study. We wanted to combine the TDD principle of writing tests before implementation with MDD. When we applied TDD, the design of the software evolved incrementally with tests, which were at the same level of abstraction as the implementation. When we applied MDD, the design of the software evolved directly with models that were at a higher level of abstraction. Thus, we discovered that TDD and MDD support the activity of design at different levels of abstraction.

Based on our observations about forward and reverse transformations, we form the opinion that depending on the objectives, RSA can be used for MDD in two different ways. If the main objective is code generation via the use of UML profiles then an MDD workflow that heavily favors forward transformations may be applied. The reverse transformation feature may not be used due to the limitation described in Section 5.3. If the main objective is design reuse and documentation without employing UML profiles for code generation then the reverse transformation feature can be used. This will encourage developers to perform activities like refactoring.

Though this thesis addresses many issues regarding the use of RSA, there are additional aspects of MDD that need to be investigated. In our case study, we briefly touched upon the use of UML Profiles. Future work can investigate in detail the use of UML profiles for capturing elaborate domain specific or design specific abstractions.

In this case study, we explored the creation of UML models for modeling in the Object-Oriented style. Another useful modeling feature in RSA is modeling database schemas. Database schema models created with RSA can be used to generate tables in the database. Additionally, RSA also provides a transformation that can transform a UML model to a relational database model. However, in order to use the output of this transformation, another IBM tool (Rational Data Architect [Rational Data Architect]) is required. A practical application of the data modeling capabilities of the IBM Rational suite of products could be another objective of future investigations.

The findings in this case study are based on the application of MDD by a single developer. Further work is required to investigate the use of RSA in a team environment. In a team environment, responsibilities could be distributed in the following manner:

- An architect or a team of architects creates a detailed design model which is implemented by a separate team of developers
- Different teams work on the same model and implementation together

If the first case is considered for large-scale projects then direct changes to the code cannot be resolved in the same manner as described in this thesis. This is because modeling and implementation are done by separate individuals, and there has to be communication between them before model and code are

synchronized. To accommodate changes from code to model, a workflow may have to be adopted by the design and development team for using RSA for MDD. The creation of such a workflow should be guided by the following questions:

- Can developers directly change the static structure (i.e., the class properties and operations) in the implementation?
- If the developer can change the static structure of classes, how does he or she communicate the change and change rationale to the architect?
- If the changes suggested by the developer are not acceptable, how can the architect communicate design alternatives?

For teams working on both model and code, a stricter workflow may be required. Such a scenario is relevant when teams are in different time zones and simultaneously working on the same module.

Therefore, further research and industrial experience (i.e., case studies) are required to understand the issues concerning model-model and model-code synchronization regarding team development². Such case studies will highlight additional limitations of MDD tools and practices, which can be improved upon. They could provide guidelines for applying MDD in team projects. It is only with such feedback and continuous improvement of MDD tools and practices that the integration of MDD into mainstream application development can become a reality.

² This latter issue is currently under investigation by one of the DSRG Ph.D. students.

References

- [Ajax 2007] AJAX: Getting Started. Online at http://developer.mozilla.org/en/docs-/AJAX:Getting_Started, Accessed on July, 2007.
- [Ambler 2004] Ambler, S. The Object Primer, Third Edition. Cambridge University Press, 2004.
- [Ambler 2007] Ambler, S. Agile Survey. Online at <http://www.ddj.com/architect-/200001986?cid=Ambysoft>, Accessed on August, 2007.
- [Astels 2003] Astels, D. Test-driven development: A practical guide. Prentice Hall, 2003.
- [Beck 1999] Beck, K. Extreme Programming Explained: Embrace Change. Addison-Wesley Professional, 1999.
- [Boehm 1987] Boehm, B. Improving Software Productivity. IEEE Computer 20, no. 9: pp 43-57, 1987.
- [Buschmann 1996] Buschmann, F., Meunier R., Rohnert H., Sommerlad P. and M.Stal. Pattern-Oriented Software Architecture, Vol. 1: A System of Patterns. Wiley, 1996.
- [Eclipse 2007] Eclipse Site. Online at <http://www.eclipse.org/>, Accessed on May, 2007.
- [Eclipse NewsList 2007] NewsList Title: Table DND Feedback. Online at <http://dev.eclipse.org-/newslists/news.eclipse.platform.swt/msg34483.html>, Accessed on July, 2007.
- [Eclipse RCP 2007] Extensible Applications. Online at <http://dev.eclipse.org/viewcvs/index.cgi-/platform-ui-home/rcp/slides/RCP.ppt?view=co#311,19,Extensible> Applications, Accessed on April, 2007.
- [Evans 2003] Evans, E. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, 2003.
- [Fowler 2000] Fowler, M. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 2000.
- [Fowler 2002] Fowler, M. Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2002.
- [Fowler 2003] Fowler, M. UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition. Addison-Wesley Professional, 2003.

- [France & Rumpe 2007] France, R., and Rumpe B. Model-driven Development of Complex Software: A Research Roadmap. International Conference on Software Engineering, 2007.
- [Gamma et al. 1995] Gamma, E., Helm R., Johnson R., and Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1995.
- [George & Williams 2003] George, B., and Williams L. An Initial Investigation of Test-Driven Development in Industry, Proceedings ACM Symposium on Applied Computing, Melbourne, FL, 2003.
- [Guttman & Parodi 2007] Guttman M., and Parodi J. Real-Life MDA: Solving Business Problems with Model Driven Architecture. Morgan Kaufmann, 2007.
- [Hailpern & Tarr 2006] Hailpern, B., and P. Tarr (2006). Model-driven development: The good, the bad, and the ugly, IBM Systems Journal, vol. 45, pp. 451-461, 2006.
- [Haskins 1997]. Haskins, C. Model Driven Development: Integrating tools with practice. IEEE Workshop on Engineering of Computer-Based Systems p. 421, 1997.
- [Hunt & Thomas 2003] Hunt, A., and Thomas D. Pragmatic Unit Testing in Java with JUnit. The Pragmatic Programmers, 2003.
- [Janzen & Saiedian 2005] Janzen, D., and Saiedian H. Test-driven development concepts, taxonomy, and future direction. IEEE Computer, Volume 38, Issue 9, 2005.
- [Java HotSpot 2007] The Java HotSpot Performance Engine Architecture. Online at <http://java.sun.com/products/hotspot/whitepaper.html>, Accessed on May, 2007.
- [Java 5 API 2004] Java 2 Platform SE 5.0 API Documentation. Online at <http://java.sun.com/~j2se/1.5.0/docs/api>, Accessed on April, 2007.
- [Kerievsky 2005] Kerievsky, J. Refactoring to Patterns. Addison-Wesley Professional, 2005.
- [Kleppe 2003] Kleppe, A., Warmer J., and Bast W. MDA Explained: The Model Driven Architecture-Practice and Promise. Model Driven Architecture-Practice and Promise. Addison-Wesley Professional, 2003.
- [Larman 2004] Larman, C. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (Third Edition). Prentice Hall PTR, 2004.

- [Mattsson 2007] Mattsson A., Lundell B., Lings B., and Fitzgerald B. Experiences from representing software architecture in a large industrial project using model driven development, ICSE, 2007.
- [McAffer & Lemieux 2005] McAffer, J., and Lemieux J. Eclipse Rich Client Platform: Designing, Coding, and Packaging Java(TM) Applications. Addison-Wesley Professional, 2005.
- [McConnell 2004] McConnell, S. Code Complete. A practical handbook of software construction. Microsoft Press, 2004.
- [Meyer 2000] Meyer, B. Object-Oriented Software Construction. Prentice Hall PTR, 2000.
- [MySQL 2007] MySQL Website. Online at <http://www.mysql.com/>, Accessed on August, 2007.
- [Nilsson 2006] Nilsson, J. 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 on April, 2007.
- [Quatrani & Palistrant 2006] Quatrani, T. and Palistrant J. Visual Modeling with IBM Rational Software Architect and UML. IBM Press, 2006.
- [Rational Data Architect 2007] Rational Data Architect. Online at <http://www-306.ibm.com/software/data/integration/rda/>, Accessed on August, 2007.
- [RSA Help 2007a] Impact of code changes on UML-to-Java transformation output. RSA 7.0.0.2 Help Documentation, 2007a.
- [RSA Help 2007b] Impact of model changes on UML-to-Java transformation output. RSA 7.0.0.2 Help Documentation, 2007b.
- [Rumbaugh 1998] Rumbaugh, J., Jacobson I., and Booch G. The Unified Modeling Language Reference Manual. Addison-Wesley Professional, 1998.
- [Selic 2003] Selic B. The pragmatics of model-driven development. IEEE Software, vol. 20, pp. 19-25, 2003.
- [Sendall & Kozaczynski 2003] Sendall, S. and Kozaczynski W. Model transformation: the heart and soul of model-driven software development. Software, IEEE. Volume 20, Issue 5, Page(s):42 – 45, 2003.

- [Swithinbank et al. 2005] Swithinbank P., Chessell M., Gardner T., Griffin C., Man J., Wylie H. and Yusuf L. Patterns: Model-Driven Development Using IBM Rational Software Architect. IBM Redbook, 2005.
- [UML 2007] Unified Modeling Language. Online at <http://www.uml.org/>, Accessed on June, 2007.
- [Weis 2003] Weis, T., Ulbrich A., and Geihs K. Model Metamorphosis. IEEE Software, Volume 20, Issue 5, Page(s): 46 – 51, 2003.

7 Appendix

This Appendix starts with RSA specific directions. Section 7.2 provides illustrations for automating the generation of the `@generated` tag in the Javadoc comments and the `//begin-user-code` and `//end-user-code` tags within methods. The purpose of these automations is to ease the work of the developer to maintain model-code synchronization while working at the implementation level.

7.1 RSA

7.1.1 Importing Java Model Libraries

If modeling of Java types is desired, then right click on the particular model (not the `.emx` file) and select *Import Model Library*. The modeling libraries for Java primitive types and Java Data types are titled *JavaPrimitiveTypes* and *Rose.JavaDataTypes* respectively.

7.1.2 View Enumeration Compartments

The Enumeration method compartments are not shown by default. To make all compartments visible, right click on the enumeration box, select *Filters → Show/Hide Compartments → All Compartments*.

7.2 Eclipse

In this customization, it is shown how to automate the creation of Javadoc templates for methods, getters and setters. We then show how to customize the tag template to be used within the method bodies while refactoring.

First, we show how to automate the addition of `@generated` tag in Javadoc comments every time a method is added. Navigate using the *Windows* menu to *Windows → Preferences → Java → Code Style* (Figure 73). Here we select the option *Automatically add comments for new methods and types*. Now the Javadoc comments that have to be generated should be placed as code templates. Navigate to *Windows → Preferences → Java → Code Style → Code Templates* (Figure 74). As seen in the right of Figure 74, select the leaf *Methods* in the *Comments* tree. We edit the template to obtain a new dialog box (Figure 75) and

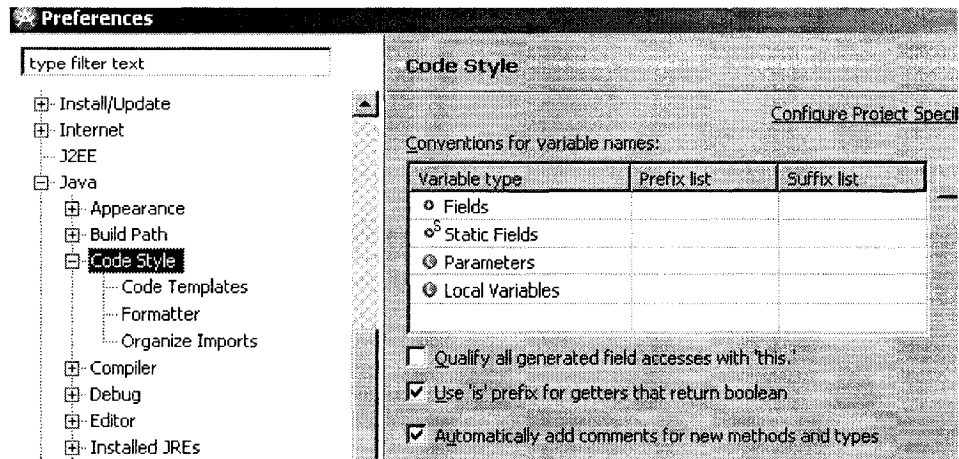


Figure 73: Code Style In Eclipse Preferences

copy paste the following template as shown in Figure 76. For getters and setters, perform the same procedure with the templates shown in Figure 77 and Figure 78 respectively.

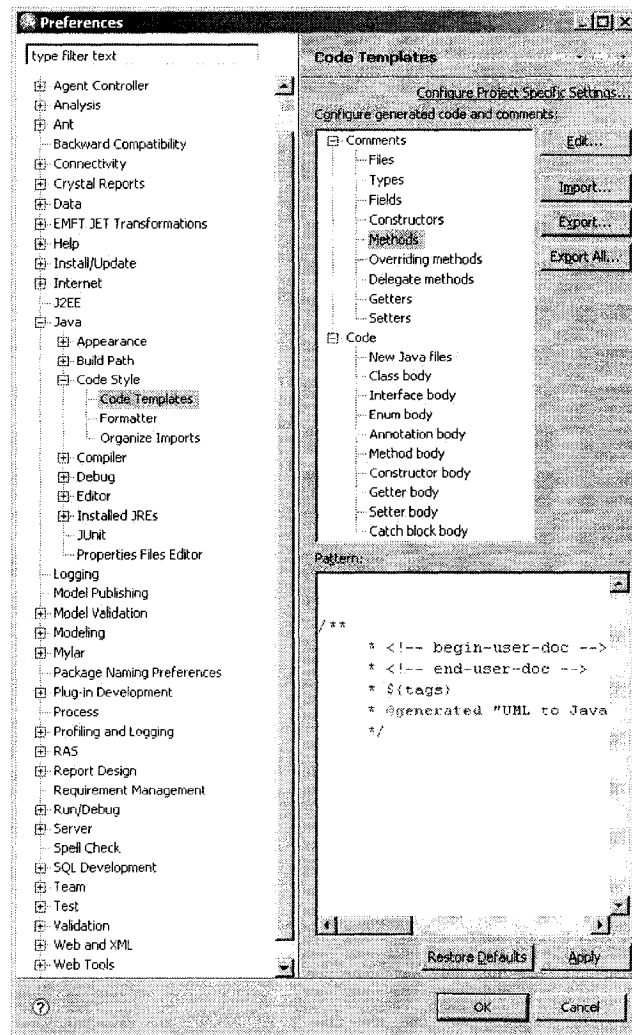


Figure 74: Snapshot of Eclipse Code Templates

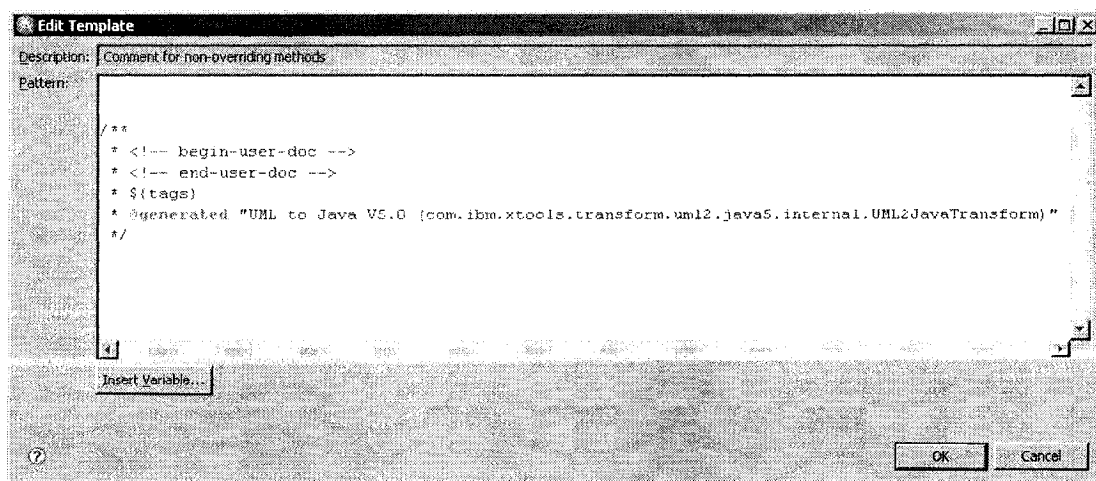


Figure 75: Comment Template for Methods

```

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

```

Figure 76: Comment Template to be pasted in the Pattern textbox for methods

```

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

```

Figure 77: Comment Template to be pasted in the Pattern textbox for getter methods

```

/**
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @param ${param} the ${bare_field_name} to set
 * @generated "UML to Java V5.0
 (com.ibm.xtools.transform.uml2.java5.internal.UML2JavaTransform)"
 */

```

Figure 78: Comment Template to be pasted in the Pattern textbox for setter methods

```

// begin-user-code
// ${todo} Auto-generated method stub
${body_statement}
// end-user-code

```

Figure 79: Code Template to be pasted in the Pattern textbox for methods

```

// begin-user-code
// ${todo} Auto-generated method stub
if(true) throw new UnsupportedOperationException("Method body not filled or wip
off");
${body_statement}
// end-user-code

```

Figure 80: Code Template to be pasted in the Methods Pattern textbox for throwing an Exception

```

// begin-user-code
return ${field};
// end-user-code

```

Figure 81: Code Template to be pasted in the Pattern textbox for getter methods

In order to add the `// begin-user-code` and `// end-user-code` within the method body, we have to navigate to the *Code* tree and then leaf *Method Body*. Similar to Figure 75, we have to copy and paste the template in Figure 79. If the guideline of throwing `UnsupportedOperationException` (See - Section 5.7.2) is to be followed, then the code template in Figure 80 should be pasted. For getters and setters, the code template to be pasted is shown in Figure 81 and Figure 82 respectively.

```
// begin-user-code
${field} = ${param};
// end-user-code
```

Figure 82: Code Template to be pasted in the Pattern textbox for setter methods

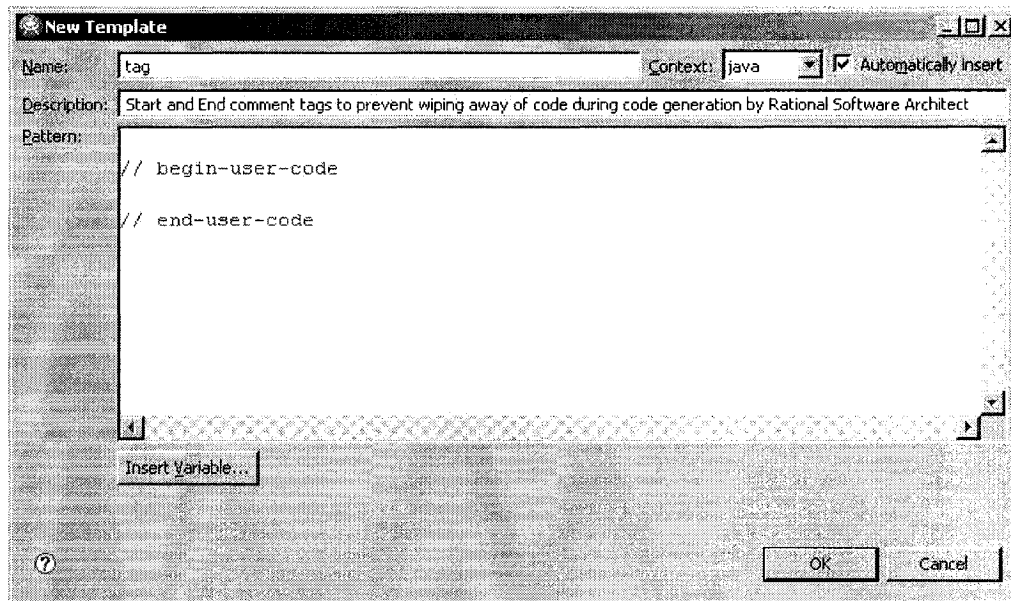


Figure 83: Creating Templates in Eclipse

The above settings take care of generating tags and Javadoc comments when creating new methods. However, for using the Extract Method feature of the Eclipse Refactoring Menu, we add a tag template. Navigate to Windows → Preferences → Java → Editor → Templates. Click on New and enter the information as shown in Figure 83. The advantage of using this template is that while working on a source file, we can use the Eclipse Content Assist feature to minimize typing. So, if a developer uses the Eclipse Content Assist feature and enters *tag*, then the `// begin-user-code` and `// end-user-code` shown in Figure 83 will be automatically generated in the selected location.

In order to generate comments for class fields or methods, place the cursor within the field declaration or the method body. Right click and select Source → Generate Element Comment. This will generate the Javadoc comments including the `@generated` tag.