



# Refactoring UML Diagrams and Models with Model-to-Model Transformations

Hafsteinn Þór Einarsson



Faculty of Industrial Engineering, Mechanical Engineering and  
Computer Science  
University of Iceland  
2011



# REFACTORING UML DIAGRAMS AND MODELS WITH MODEL-TO-MODEL TRANSFORMATIONS

Hafsteinn Þór Einarsson

60 ECTS thesis submitted in partial fulfillment of a  
*Magister Scientiarum* degree in Software Engineering

Advisors

Helmut Wolfram Neukirchen

Ebba Þóra Hvannberg

Faculty Representative

Jóhann Pétur Malmquist

Faculty of Industrial Engineering, Mechanical Engineering and  
Computer Science

School of Engineering and Natural Sciences

University of Iceland

Reykjavik, May 2011

Refactoring UML Diagrams and Models with Model-to-Model Transformations  
Refactoring UML models and diagrams  
60 ECTS thesis submitted in partial fulfillment of a M.Sc. degree in Software Engineering

Copyright © 2011 Hafsteinn Þór Einarsson  
All rights reserved

Faculty of Industrial Engineering, Mechanical Engineering and  
Computer Science  
School of Engineering and Natural Sciences  
University of Iceland  
Hjarðarhagi 2-6  
107, Reykjavík, Reykjavík  
Iceland

Telephone: 525 4000

Bibliographic information:

Hafsteinn Þór Einarsson, 2011, Refactoring UML Diagrams and Models with Model-to-Model Transformations, M.Sc. thesis, Faculty of Industrial Engineering, Mechanical Engineering and  
Computer Science, University of Iceland.

Printing: Háskólaprent, Fálkagata 2, 107 Reykjavík  
Reykjavík, Iceland, May 2011





# Abstract

Software is becoming increasingly important in everyday life and is becoming increasingly complex as well. Techniques have been developed to reduce the complexity, e.g. abstract modelling, model-driven development and refactoring code structure. Refactoring is a systematic approach to restructure code to make it simpler without changing its behaviour. Refactoring has been well investigated and is proven for programming languages but is still a developing concept in model-driven development. The refactoring process has already been applied to UML models in earlier work but the focus has only been on the elements of a UML model without updating the typically associated diagrammatic representation. In this thesis, automated refactorings are developed for restructuring UML activity models together with their diagrammatic representation using the QVT operational transformation language for transforming UML models and diagrams created with the Papyrus UML editor.





# Ágrip

Hugbúnaður verður æ mikilvægari í daglegu lífi og einnig flóknari um leið. Ýmsar aðferðir hafa verið þróaðar til að minnka flækjustigið, t.d. líkanagerð, líkanadrifin þróun og endurþáttun kóða (e. refactoring). Endurþáttun snýst um að gera uppbyggingu kóða einfaldari án þess að hafa áhrif á hegðun hans. Endurþáttun hefur verið mikið rannsökuð og prófuð fyrir forritunarmál en er enn í þróun sem hugtak fyrir líkanadrifna hugbúnaðarþróun. Í áður útgefnu efni hefur endurþáttun á líkönum verið skilgreind fyrir UML líkөн en aðeins hefur verið einblínt á líkөнin sjálf án tillits til grafískrar uppsetningar þeirra. Í þessari ritgerð eru skilgreind dæmi um sjálfvirka endurþáttun á UML aðgerðaritum (e. activity diagram) sem taka bæði tillit til líkansins og grafískrar uppsetningar þess með því að nota QVT líkanaumbreytingarmálið til að umbreyta UML ritum sem búin eru til í Papyrus UML ritlinum.



# Preface

This M.Sc. project was carried out at the Faculty of Industrial Engineering, Mechanical Engineering and Computer Science at the University of Iceland.

Special thanks goes to my supervisor, Associate Professor Helmut Neukirchen at the University of Iceland, his dedication and guidance were crucial to the project and he always had time to help. I also greatly appreciate Ebba Þóra Hvannberg's efforts as secondary supervisor as well as Jóhann Malmquist's contribution as faculty representative. I must also thank my partner Eva for her support and encouragement throughout the project. Finally I would like to thank my good friend Oddgeir Guðmundsson for all his good advice and of course Böðvar Hlöðversson for always being an inspiration.

Reykjavík, May 2011.

Hafsteinn Þór Einarsson



# Contents

<b>List of Figures</b>	<b>xv</b>
<b>Listings</b>	<b>xvii</b>
<b>1. Introduction</b>	<b>1</b>
<b>2. Foundations</b>	<b>3</b>
2.1. Refactoring . . . . .	3
2.2. The Unified Modeling Language . . . . .	4
2.2.1. The UML infrastructure . . . . .	5
2.2.2. The UML superstructure . . . . .	6
2.2.3. The Object Constraint Language . . . . .	9
2.2.4. XML Metadata Interchange . . . . .	10
2.2.5. The UML Diagram Interchange . . . . .	10
2.3. The Eclipse Project . . . . .	10
2.3.1. Plugins . . . . .	10
2.3.2. The Eclipse Modeling Framework . . . . .	11
2.3.3. Papyrus . . . . .	14
2.3.4. Model-To-Model . . . . .	17
2.3.5. Operational QVT . . . . .	17
<b>3. Related Work</b>	<b>19</b>
3.1. UML refactoring . . . . .	19
3.2. Tool support . . . . .	21
3.3. Discussion . . . . .	22
<b>4. UML Refactorings</b>	<b>23</b>
4.1. Merge actions . . . . .	23
4.1.1. Motivation . . . . .	24
4.1.2. Refactoring mechanics . . . . .	24
4.2. Divide action . . . . .	24
4.2.1. Motivation . . . . .	25
4.2.2. Refactoring mechanics . . . . .	25
<b>5. Tool Implementation</b>	<b>27</b>
5.1. Model-based vs. low-level transformations . . . . .	27

5.2.	Tools . . . . .	27
5.3.	Challenges . . . . .	28
5.3.1.	Documentation . . . . .	28
5.3.2.	Finding selected objects in the model . . . . .	29
5.3.3.	Creating a new object in the model . . . . .	29
5.3.4.	Accessing coordinates of diagram elements . . . . .	29
5.3.5.	Transforming two models . . . . .	29
5.4.	Metamodel classes . . . . .	30
5.4.1.	Notation metamodel classes . . . . .	30
5.4.2.	UML metamodel classes . . . . .	31
5.5.	Plugin architecture . . . . .	31
5.5.1.	Workflow . . . . .	33
5.6.	Blackboxing library . . . . .	35
5.6.1.	isSelectedObject() . . . . .	36
5.6.2.	moveElement() . . . . .	36
5.6.3.	setElementID() . . . . .	37
5.7.	Executing the transformation . . . . .	37
5.7.1.	Invoking a QVTO transformation with Java . . . . .	38
5.7.2.	Adding a context menu entry . . . . .	42
5.8.	Merge actions refactoring . . . . .	44
5.8.1.	The getObjects() query . . . . .	45
5.8.2.	The merge() mapping . . . . .	46
5.8.3.	UML model element mappings . . . . .	48
5.9.	Divide action refactoring . . . . .	49
5.9.1.	The getObjectToDivide() query . . . . .	50
5.9.2.	The addNode() mapping . . . . .	51
5.9.3.	The addEdge() mapping . . . . .	52
5.9.4.	The addNodeAndEdge() mapping . . . . .	53
5.9.5.	The setSource() mapping . . . . .	54
<b>6.</b>	<b>Evaluation</b>	<b>55</b>
6.1.	The example activity diagram . . . . .	55
6.2.	The Merge actions refactoring . . . . .	60
6.2.1.	The UML model comparison . . . . .	64
6.2.2.	The Notation model comparison . . . . .	66
6.3.	The Divide actions refactoring . . . . .	68
6.3.1.	The UML model comparison . . . . .	71
6.3.2.	The Notation model comparison . . . . .	74
6.4.	The finishing touches . . . . .	76
6.5.	Discussion . . . . .	78
<b>7.</b>	<b>Conclusion</b>	<b>79</b>
7.1.	Summary . . . . .	79
7.2.	Outlook . . . . .	79

<b>Acronyms</b>	<b>81</b>
<b>Bibliography</b>	<b>83</b>
<b>A. The UML refactoring plugin</b>	<b>87</b>
A.1. Dependencies . . . . .	87
A.2. Obtaining the code from an SVN repository . . . . .	90
A.3. Installing the plugin . . . . .	94
A.4. Using and debugging the plugin . . . . .	95





# List of Figures

2.1. An example of the four-layer metamodel hierarchy [32] . . . . .	6
2.2. A simple activity diagram [19] . . . . .	8
2.3. The Notation metamodel . . . . .	13
2.4. An Activity diagram created with Papyrus . . . . .	15
2.5. The relevant parts of Figure 2.4 in more detail . . . . .	16
4.1. The Merge actions refactoring . . . . .	23
4.2. The Divide actions refactoring . . . . .	25
5.1. The structure of the plugin . . . . .	32
5.2. The workflow of the refactoring transformation . . . . .	34
6.1. Creating a new project in Eclipse . . . . .	56
6.2. Creating a new Papyrus project in Eclipse . . . . .	56
6.3. Naming the Papyrus project . . . . .	57
6.4. Selecting the diagram type . . . . .	57
6.5. Adding an activity diagram to the project. . . . .	58
6.6. The Papyrus workbench . . . . .	59
6.7. The example activity diagram . . . . .	60

## LIST OF FIGURES

6.8. The Merge actions refactoring selected . . . . .	61
6.9. A dialog telling the user that the model resource has changed. . . . .	62
6.10. The diagram after the merge actions refactoring. . . . .	63
6.11. The Divide actions refactoring selected . . . . .	69
6.12. The diagram after the divide actions refactoring. . . . .	70
6.13. The final model . . . . .	77
A.1. Installing the Eclipse Modeling Discovery UI . . . . .	88
A.2. Selecting the model installation UI . . . . .	89
A.3. Installing QVTO and Papyrus UML . . . . .	90
A.4. Creating a new SVN project . . . . .	91
A.5. Selecting the head revision . . . . .	92
A.6. Finding the projects in the SVN repository . . . . .	93
A.7. Confirming the projects found in the SVN repository . . . . .	93
A.8. Selecting the plugins for export . . . . .	95
A.9. The plugins debugged in a new eclipse instance . . . . .	96
A.10. Creating a debug configuration for a QVTO transformation . . . . .	97
A.11. Finding a Shape object's GUID in a Notation model . . . . .	97
A.12. Setting the value of a configuration property . . . . .	98

# Listings

2.1. An example QVTO transformation . . . . .	18
5.1. Making a blackboxing library available in plugin.xml . . . . .	35
5.2. The isSelectedObject() blackboxing operation . . . . .	36
5.3. The moveElement blackboxing operation . . . . .	36
5.4. The setElementID() blackboxing operation . . . . .	37
5.5. The run() method in RefactoringInvocation.java . . . . .	38
5.6. The getSelection() method in MergeActions.java . . . . .	41
5.7. The initTransformation() method in MergeActions.java . . . . .	41
5.8. The setProperties() method in MergeActions.java . . . . .	42
5.9. Adding a context menu entry in plugin.xml . . . . .	43
5.10. The MergeActions.qvto transformation . . . . .	44
6.1. The UML model before refactoring . . . . .	65
6.2. The resulting UML model after the Merge actions refactoring . . . .	66
6.3. The Notation model before the refactoring . . . . .	67
6.4. The resulting Notation model after the Merge actions refactoring . .	68
6.5. The UML model before refactoring . . . . .	72
6.6. The resulting UML model from the Divide actions refactoring . . . .	73
6.7. The Notation model before refactoring . . . . .	74
6.8. The resulting Notation model from the Divide action refactoring . . .	75



# 1. Introduction

Software is an important part of everyday life, and is becoming increasingly important every day. As software becomes ever more present in the world, it also becomes more complex. When complexity increases it gets harder to keep an overview of the whole software. However, some techniques have been developed and utilized to reduce the complexity, e.g. making abstract models of the software, model-driven development and refactoring the code structure.

The term refactoring was coined by William Opdyke [34] in 1992 and further defined by Martin Fowler [18] in 1999. Refactoring is defined as a process of restructuring software to improve quality and readability without changing its behaviour. This process can be automated through the use of software tools which have already been implemented for most popular programming languages.

Model-driven development is a discipline in software engineering where the focus is on designing and creating models instead of developing the software logic in code, often by using the Unified Modeling Language (UML) [29, 32]. UML models can be created graphically using UML editors that turn the graphical elements into an underlying UML model representation. These UML models can then e.g. be used as input for code generators. While the implementation of automatic refactorings has already been done on source code level, the shift to do the programming on the model level also requires refactoring at the model level. However, even though the refactoring of UML models has been studied using model-to-model transformations (M2M) [2, 36], a thorough search revealed that almost no tools exist for automating model refactoring. Refactoring models is also different from refactoring code, because models often have a diagrammatic representation so the layout of elements has to be taken into account. The main focus of the research that has already been done on model refactoring has been on the underlying model but not on the diagram itself. This means that aspects of refactoring the graphical models have not been addressed, e.g. the placement of new elements that are added to a diagram in the process of refactoring.

The aim of this thesis is to develop and implement automatic refactorings for UML models and diagrams which refactor the UML model itself and the diagram model as well. Example refactorings, that refactor activity diagrams, are developed as an Eclipse IDE [17] plugin which works with the Eclipse Papyrus UML editor [14] and

## *1. Introduction*

utilizes the QVTO transformation language [15] to implement the UML model and diagram refactorings using a model-based approach.

The structure of this thesis is as follows: Chapter 2 describes the foundations of the technologies used in the thesis and covers the refactoring process, the UML, the QVT transformation language and the Eclipse project. In chapter 3 the work that has already been done on refactoring models is examined. Chapter 4 outlines the refactorings developed in an abstract way and shows the steps needed in the refactoring process. In chapter 5, where the main contribution of the thesis resides, the implementation of the refactorings is explained in detail, both the support structure and the QVT transformations. In chapter 6 the refactoring implementation is evaluated by its application to an example diagram and the resulting models compared to the original models. Chapter 7 concludes the thesis with a summary and an outlook.

## 2. Foundations

In this chapter the main fundamentals behind the thesis are presented. Section 2.1 presents the basics of code refactoring and its history. In section 2.2 the basics of the Unified Modeling Language are presented as well as some related technologies, e.g. the Meta-Object Facility (MOF) and the Query/View/Transformation (QVT) standard for model-to-model transformations. Section 2.3 introduces the Eclipse project and some of its extensions and frameworks.

### 2.1. Refactoring

Software is known to become less maintainable over time, as code is added to satisfy new requirements or changed to fix errors. This can lead to a gradual diversion from the original design of the software which can make changes harder to make. David Parnas refers to this as software aging [35]. To mend this problem one has to restructure the software to remove clutter and make it more readable. This restructuring process has been named refactoring.

Martin Fowler defines refactoring in the following way:

“**Refactoring** (noun): A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.” [18]

In other words, refactoring is generally used to simplify code structure, which can make code easier to read and ease maintainability. The process of refactoring consists of a sequence of systematic mechanical steps, where each step is a transformation of the code structure which preserves its behavior. Fowler refers to these steps as the mechanics of a refactoring which are systematic, predictable steps to keep with the "never change a running system" attitude. Often the need to refactor code arises from code smells, where a pattern or symptom can be recognized in code which can indicate a potential problem, e.g. when a class has grown too large to be easily maintained or changed. An example of a refactoring to address this code smell is the *Extract Class* [18] refactoring. Fowler describes the manual steps included in the process:

## 2. Foundations

1. Decide how to split the responsibilities of the class
2. Create a new class to express the split-off responsibilities
3. Make a link from the old to the new class
4. Move chosen fields to new class
5. Compile and test after each move
6. Review and reduce the interfaces of each class
7. Decide whether to expose the new class

Fowler emphasizes greatly the need to run automatic unit tests after each step in the manual refactoring process to minimize the risk of introducing new bugs or changing the behavior in any way. However, some refactorings have been automated in a number of tools. In these cases automatic tests may not be necessary. William Opdyke argues that it is not possible to automate every refactoring, but when it is possible the tool should make sure that defects are not introduced into a program [34].

## 2.2. The Unified Modeling Language

The Unified Modeling Language (UML) is a standardized modeling language created by the Object Management Group [33]. The first version of the language was published as a standard in 1997. The UML is intended for designing and describing object-oriented software systems. The UML specification has four major parts:

- Infrastructure
- Superstructure
- The Object Constraint Language (OCL)
- The UML Diagram Interchange



### 2.2.1. The UML infrastructure

The main component of the UML infrastructure is the Meta-Object Facility (MOF) [32], which is a metamodeling tool. There are two variants of the MOF [30]:

**Essential MOF (EMOF)** a basic representation of the MOF, only allows simple metamodels

**Complete MOF (CMOF)** a fully featured representation of the MOF

The MOF has a four layered architecture with itself in the top layer (M3) as a meta-meta model which conforms to itself. The M2 layer consists of a metamodel for the UML language which is defined using the MOF, the M1 layer consists of instances of user models, e.g. a UML class diagram and the M0 bottom layer consists of run-time instances. This is illustrated in fig. 2.1.

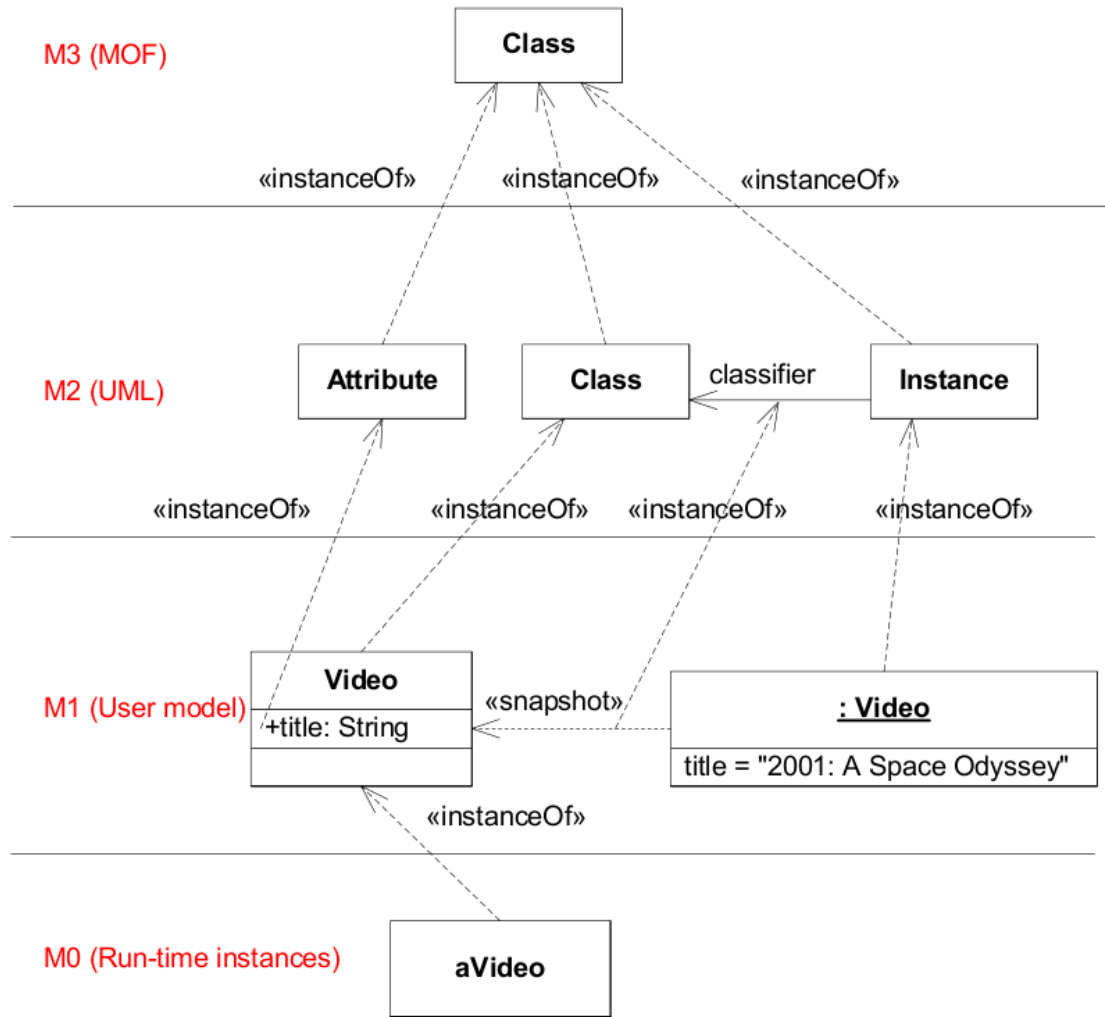


Figure 2.1: An example of the four-layer metamodel hierarchy [32]

### 2.2.2. The UML superstructure

The UML superstructure [29] defines the semantics for diagrams and their model elements. Note that in this thesis the following distinction is made between a model and a diagram, because in practice the term diagram is often used to describe a diagram and a model:

**Model** A model is the description of a set of elements and their relationships but not their layout or representation in a diagram.

**Diagram** A diagram is the diagrammatic representation of a model.

It is possible to have a model without a diagrammatic representation, such a model could be used as input for a code generator. Additionally a diagram may not have a corresponding model, e.g. when a diagram is drawn on paper. A UML editor typically allows the user to draw diagrams besides the underlying model. However, as the underlying model does not contain any graphical information, e.g. coordinates, it is not possible to create a diagram from a model.

The UML proposes a few kinds of models and diagrams, the following can be considered the most common [19]:

- Structural diagrams
  - *Class diagram*, describes classes, their operations and attributes and the relationships between them
  - *Component diagram*, describes components and their dependencies
  - *Package diagram*, describes how systems are broken down into packages, most often a group of classes
- Behavior diagrams
  - *Activity diagram*, describes procedural logic and work flow, see Figure 2.2
  - *Use case diagram*, describes how users interact with the system
  - *State machine diagram*, describes the behavior of a system and how it moves between states
  - Interaction diagrams
    - \* *Communication diagram*, describes the interaction between objects
    - \* *Sequence diagram*, describes how processes send information between each other and in what order

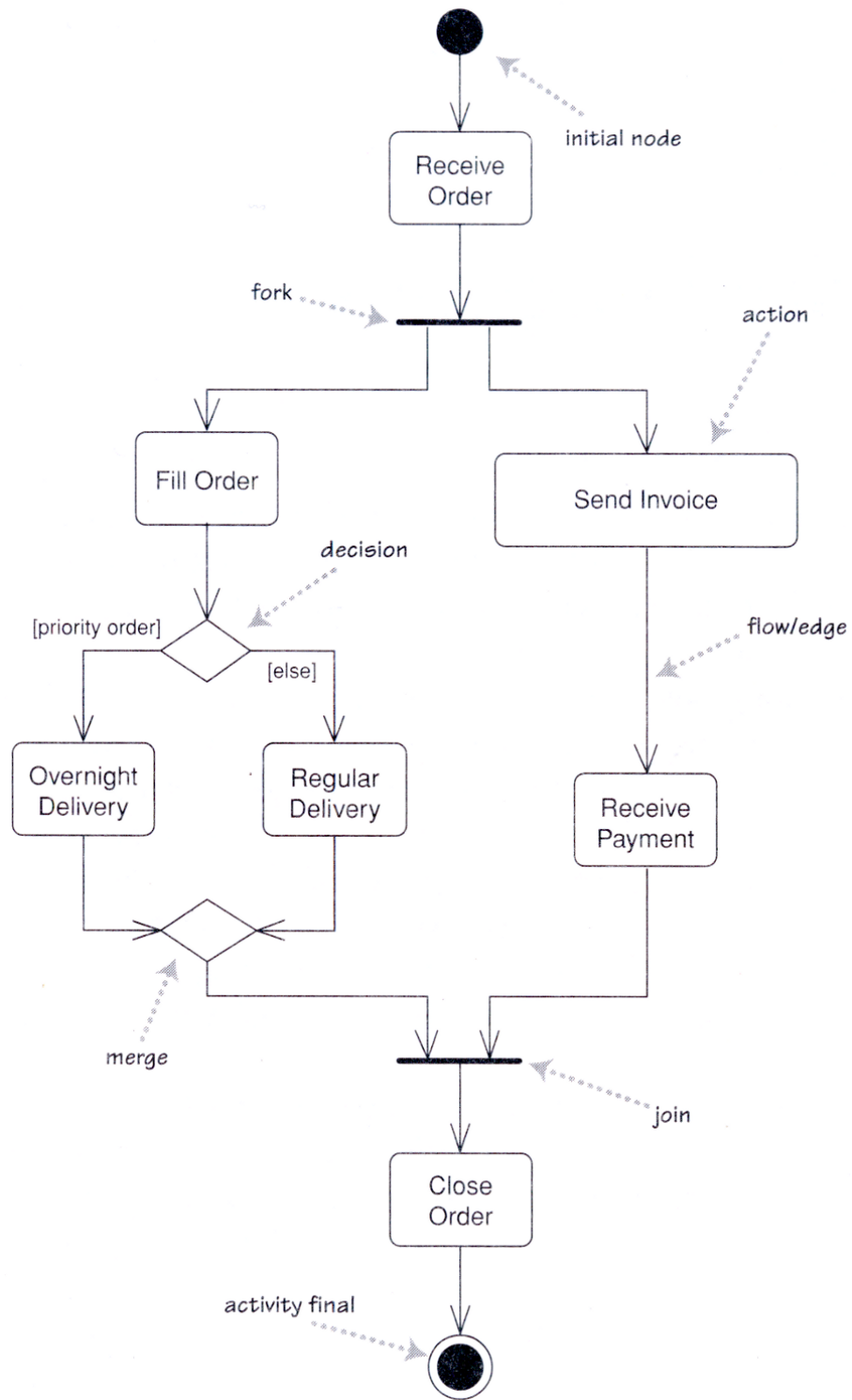


Figure 2.2: A simple activity diagram [19]

### 2.2.3. The Object Constraint Language

The Object Constraint Language (OCL) [27] is an extension language for MOF metamodels to describe formal constraints or expressions which can not be communicated through diagrams. It was originally designed as an extension for the UML but is now intended for every MOF compliant metamodel.

#### Query/View/Transformation

The Query/View/Transformation language (QVT) [26] is a standard for transforming models. It is developed by the OMG and can be used to transform any MOF compliant model into any MOF compliant model. The QVT specification integrates the OCL and adds three languages on top:

**QVT Core** The OMG defines the QVT core language as a

“small model/language which only supports pattern matching over a flat set of variables by evaluating conditions over those variables against a set of models.” [26]

QVT core is a declarative language and it only has an abstract syntax.

**QVT Relations** The relations language is declarative as core, but offers more complex object pattern matching and has a concrete syntax.

**QVT Operational mappings** The operational mappings extends both relations and core but it is imperative and offers a procedural concrete syntax, so it might be easier to grasp for programmers who only have experience with procedural languages.

QVT also adds a mechanism called Black Box implementations which adds the possibility of calling external libraries that are implemented in other languages. This can be useful, i.e. when the domain has complex algorithms that can not be expressed in OCL [26]. An example of a QVT operational transformation is shown in section 2.3.5.

### 2.2.4. XML Metadata Interchange

The XML Metadata Interchange (XMI) [28] is a standard format for exchanging information about MOF compliant models using the Extensible Markup Language (XML). It includes information about elements in a model and their relationships. The diagrammatic representation or layout is not contained in XMI because the MOF does not contain information about layout.

### 2.2.5. The UML Diagram Interchange

The UML Diagram Interchange (UMLDI) is a standard format for exchanging UML diagrams between different UML software tools, including information regarding diagrams and their layout [31]. This format is based on the XML Metadata Interchange (XMI) standard described in section 2.2.4 but adds a specification for diagrammatic information which is not possible with the XMI. The UMLDI is quite beneficial as it allows users to interchange their models between programs, but it is not very widely used in practice. However, at least one tool claims conformance to the format [44].

## 2.3. The Eclipse Project

The Eclipse project was originally created in 2001 by IBM [17]. It is an open development platform comprised of frameworks which can be extended with plugins. The most known project within the Eclipse project is the Eclipse integrated development environment (IDE). It was originally designed for the Java programming language but it also has extensions and plugins which support other popular programming languages.

### 2.3.1. Plugins

An important part of the Eclipse environment is the ability to extend it with plugins that add functionality to it. The Eclipse IDE has a built in mechanism for easing the development of a plugin, it is e.g. possible to run the plugin in another instance of Eclipse for testing and debugging purposes and packaging the plugin can be mostly done automatically.

### Extension points

Extension points is a part of the plugin mechanism in Eclipse which can be used to define a point in one plugin to which another plugin contribute. An example of this is a mechanism for easily adding menu entries and other simple functions to the user interface.

### 2.3.2. The Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) is a framework for Java code generation based on models [41]. It is a unification of three technologies: XMI, UML and Java. A model can be defined in any of these forms, and then the other can be automatically generated. An example of this would be to create a UML model in a diagram editor which can save the model in a UML format. With that model EMF can automatically generate both an XMI model and Java code which represent the model, and changes to either generated model can be propagated to the other generated model and the original model.

### Ecore

The EMF provides Ecore, an EMF model which closely resembles the EMOF [41]. Ecore is a metamodel for itself so it is a meta-metamodel. It is intended for creating metamodels for domain models and adds runtime support for the models including [7]:

- Change notification
- XMI serialization for model persistence
- A reflective Application Programming Interface (API) for manipulating EMF objects

### UML2

UML2 is a part of the Model Development Tools Eclipse project (MDT) [12]. It is an EMF-based implementation of the UML 2 metamodel by OMG, specified as an Ecore metamodel. It also provides an XMI schema to ease interchange between programs focusing on the model but not the diagrammatic representation.

## *2. Foundations*

### **The Graphical Editing Framework**

The Graphical Editing Framework (GEF) is a generic Eclipse framework which adds the possibility to create graphical editors for diagrams [8]. Many graphical editors for specific diagrams are based on the GEF.

### **The Graphical Modeling Project**

The Graphical Modeling Framework (GMP) is an Eclipse project which builds upon the EMF and the GEF to provide a runtime infrastructure for the ability to create graphical diagram editors which work with concrete EMF models [9].

### **The Notation metamodel**

The GMP includes an EMF metamodel for persisting diagram information independent from the domain model [10], called the Notation metamodel. The Notation metamodel structure is shown in Figure 2.3.



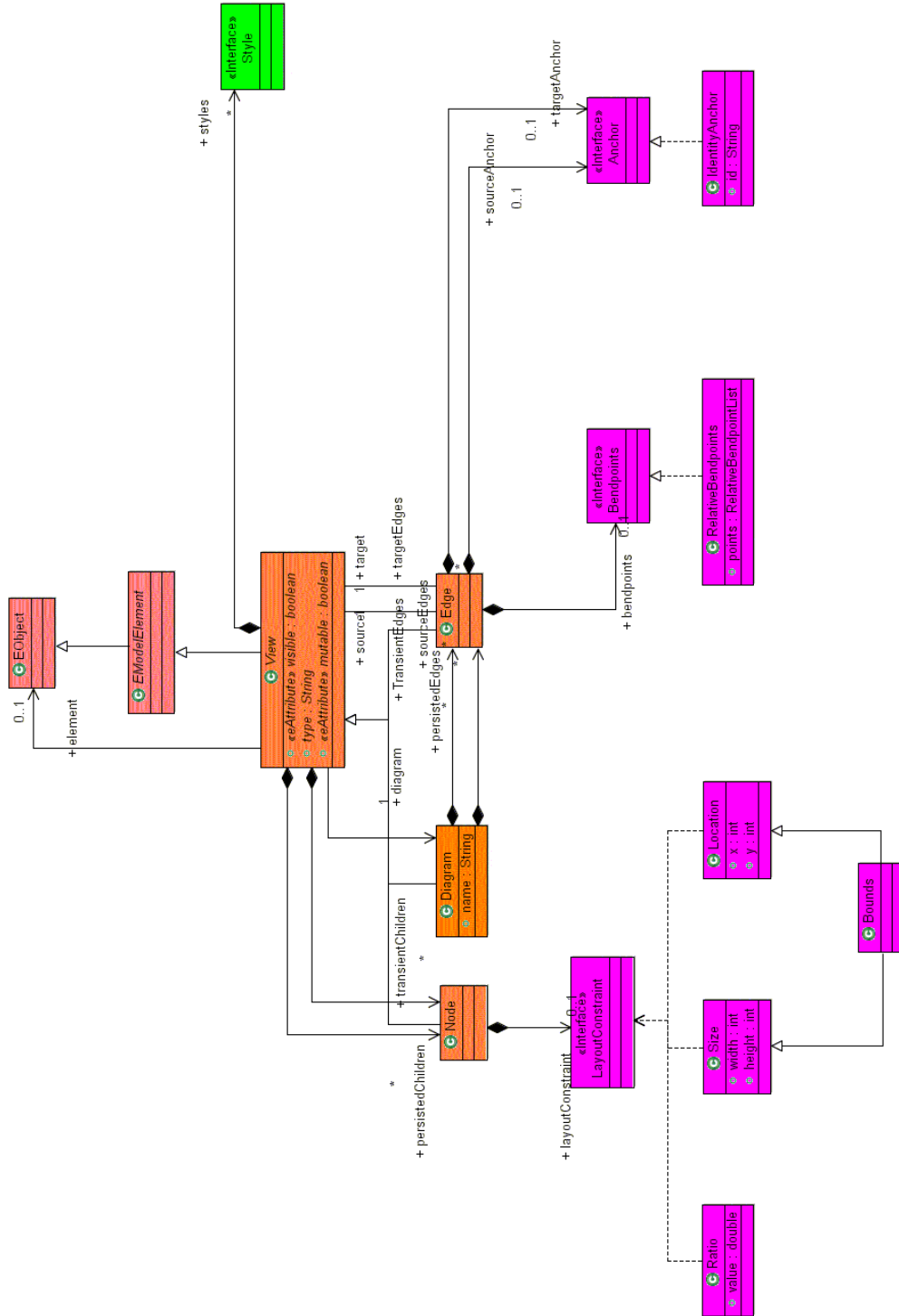


Figure 2.3: The Notation metamodel

## 2. Foundations

The most important classes in the Notation metamodel in Figure 2.3 are the *View*, *Node* and *Edge* classes. A *View* is a general class of which *Edge* objects and *Node* objects are instances. A *Node* has attributes, i.e. size and position and edges have attributes which include the corners where an edge bends, called bendpoints, and an anchor which describes where it connects to a node.

### 2.3.3. Papyrus

Papyrus is a GMP based project to provide Eclipse with a fully fledged UML editor along with supporting other related languages such as SysML and MARTE [14]. It stores diagram information in a notation model and model specific information in an EMF model. In the case of UML the information is stored in a UML2 model. Papyrus currently supports the most popular [19] UML diagrams:

- Activity diagram
- State machine diagram
- Communication diagram
- Sequence diagram
- Use case diagram
- Class diagram
- Package diagram
- Composite structure diagram

When a new diagram is created with Papyrus, three files are created based on a name given by the user. If the diagram is named *ActivityExample*, the following XMI compliant files are automatically created:

**ActivityExample.di** The *.di* file holds metadata about the diagram and a path to the Notation model.

**ActivityExample.notation** The *.notation* file stores information about the diagram elements, e.g. size, position and a link to the element in the UML model.

**ActivityExample.uml** The *.uml* file keeps all information about the underlying UML model elements.

The diagram and the associated files Papyrus creates can be seen in Figure 2.4 and the relevant parts can be seen in more detail in Figure 2.5.

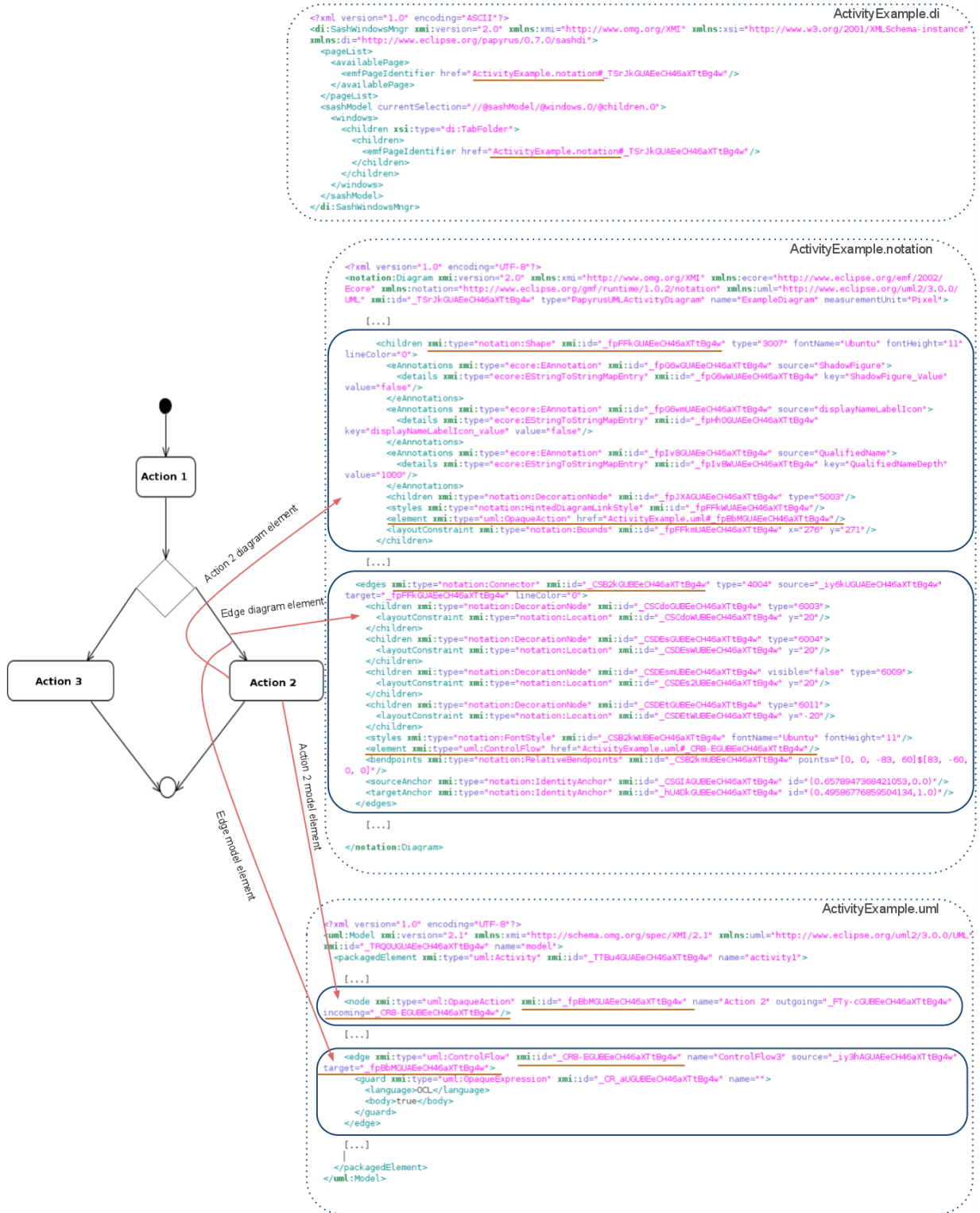


Figure 2.4: An Activity diagram created with Papyrus

## 2. Foundations



Figure 2.5: The relevant parts of Figure 2.4 in more detail

Figure 2.4 illustrates a part of the files associated with an activity diagram created with Papyrus. The figure shows a very simple activity diagram and highlights the representations of an action and a control flow edge, with some important parts underlined. The ActivityExample.di file contains a link to the ActivityExample.notation file and its globally unique identifier (GUID), which is a randomly generated string of characters as seen in the underlined parts of the file.

In the file ActivityExample.notation in Figure 2.5 two important objects can be seen. The first one is an object with the type *notation:Shape* and represents the *Action 2* element in the diagram. The second object has the type *notation:Connector* and

represents the incoming edge to *Action 2*. Both of these objects hold the *element* property which is a link to the UML model element the diagram element represents. The link points to the GUID of the element in the ActivityExample.uml file, and it can be seen that elements with these GUIDs exist there. The *notation:Shape* object also contains a *layoutConstraint* property which holds information about the position of the diagram element.

In the ActivityExample.uml file in Figure 2.5 the object of the type *uml:OpaqueAction* represents the *Action 2* element as can be seen by the name property. The *incoming* property holds the GUID of the control flow edge discussed here. The edge object of the type *uml:ControlFlow* in turn has a target property which holds the GUID of the *Action 2* element.

#### 2.3.4. Model-To-Model

Model-To-Model (M2M) is a subproject of the Eclipse Modeling Project (EMP). It provides three transformation engines for model-to-model transformations [11]:

**Declarative QVT** is based on the OMG's QVT core and relations languages, but is still work in progress.

**Operational QVT** is based on the OMG's QVT operational package mappings and provides a procedural language with concrete syntax. It is used in the implementation in this thesis and is explained in more detail in section 2.3.5.

**The ATL Transformation Language (ATL)** is a transformation language with an IDE and debugger built on top of the Eclipse platform.

#### 2.3.5. Operational QVT

Operational QVT (QVTO) is an open source implementation of the imperative QVT standard from the OMG [11]. It provides a transformation engine that can perform model-to-model transformations on any Ecore compliant model when provided with a metamodel for both input and output models. There are two types of transformations:

**Exogenous transformation** The target metamodel is different from the source metamodel.

**Endogenous transformation** The source and target metamodels are the same.

## 2. Foundations

Listing 2.1 shows an example of a very simple exogenous QVTO transformation:

```
1 transformation Book2Publication(in bookModel:BOOK, out pubModel:PUB);
2
3 main() {
4   bookModel.objectsOfType(Book)->map book_to_publication();
5 }
6
7 mapping Book::book_to_publication() : Publication {
8   title := self.title;
9   nbPages := self.chapters->nbPages->sum();
10 }
```

*Listing 2.1: An example QVTO transformation*

In this example there are two metamodels. The Ecore metamodels are not shown, but they can be summarized as follows:

**Book** A book has a title and sequence of chapters, where each chapter has a title and a number of pages

**Publication** A publication has a title and a number of pages

The transformation takes in a Book input model and returns a Publication model. The *main* function in lines 3-5 is the entry point to the transformation, that's where the execution starts. There each book in the input model is transformed to a publication with the *book\_to\_publication* mapping, where the actual transformation takes place. In the mapping, each book object is transformed to a publication object. Each resulting publication object gets the title of the book and the number of pages is calculated from the sum of the page numbers of all the chapters from the book object.

### Java Black-boxing

QVTO provides a way to call external Java libraries with QVT's blackboxing mechanism. This can be beneficial when complex calculations are needed or when the metamodel does not provide setting or fetching attributes of an element. An example of this is shown in chapter 5.

## 3. Related Work

In this chapter earlier work on the refactoring of models will be presented. The work is mostly focused on UML models because they are a widely used standard in the software world. Most of the work that has been done in regards to refactoring models has been on the domain model level and not on the diagram layout level.

### 3.1. UML refactoring

Gerson Sunyé et al. [42] propose a set of refactorings on UML models, specifically class models and state machine models. They present a catalog of transformations and divide them into three categories:

- Add feature/association
- Remove feature/association
- Move element

Additionally, OCL pre- and postconditions are defined for each state machine refactoring to show that the behaviour of the machine is preserved.

France and Bieman [38] identify two classes of model transformations:

**Vertical transformation** results in a model at a different level of abstraction

**Horizontal transformation** results in a model at the same level of abstraction

They propose model transformations based on design patterns [20]. A design pattern-based transformation is decided by role models, that is, models that define the design pattern precisely.

Marko Boger and Per Fragemann [4] implement a refactoring browser and include within it a few refactorings for state machine models and two for activity models:

### 3. Related Work

**Make actions concurrent** Creates a split and join pseudostate for concurrent activities and moves actions between them

**Sequentialize concurrent actions** Removes a pair of split and join pseudostates and makes actions in between them sequential.

The refactorings are implemented as a plugin to the Poseidon UML tool [21] but the models are not based on an OMG standard.

Dave Astels [1] presents a few examples of smells in UML class models and implements refactorings to address them. He reasons that sometimes smell detection can be easier in UML as it has a different view at the structure of the program than the code itself. He adds that it would be beneficial to refactor the models to react to the smells as tools will increasingly be able to keep the code in sync with the models.

Kexing Rui and Greg Butler [39] provide a catalog of use case model refactorings. The metamodel is based on Regnell's use case model [37] with a few modifications, but should mostly be compatible with the UML use case model.

Ivan Porres [36] presents an action language which resembles OCL that is capable of model transformation. It is a rule-based language which means that when a set of preconditions are satisfied, a sequence of transformations are executed. He provides a few refactorings for class models and state machine models and implements them in an experimental tool. He reasons that adding new elements in a graphical diagram can be problematic because the position can not be easily inferred.

Slaviša Marković and Thomas Baar [2] develop a few refactorings for class models using QVT. They focus on OCL annotated models so that any changes made by refactoring a model are automatically reflected in OCL constraints. However, they only focus on the underlying UML model and not the graphical representation.

Łukasz Dobrzański [5] develops a catalog of refactorings for executable UML, that is, UML models that are precisely specified and thus can be compiled to executable code. He focuses on refactoring class models and defines the refactorings in OCL. He does not focus on the graphical representation of the diagrams, only the underlying model.

Graph transformation theory defines how graphs can be transformed to make new graphs. UML models can be seen as graphs [23] so graph transformation theory can be useful for executing automatic UML refactorings. Enrico Biermann et al. [3] present a few refactorings on state machine models in EMF using graph transformations. They use the method to validate model transformations to ensure their correctness and preservation of functional behaviour. The focus is only on the underlying EMF model and not its graphical representation.



Alessandro Folli and Tom Mens [6] develop the Attributed Graph Grammar System (AGG) using graph transformations to execute refactorings on UML models. They focus on class models and state machine models and define a metamodel similar to the UML metamodel as a type graph. There are some limits to this approach however, because the type graph can only represent a simplified version of the UML metamodel.

## 3.2. Tool support

There exist many tools for drawing UML diagrams. However not many of them give the user a possibility of automatically refactoring. MagicDraw [25] offers two refactorings:

**Convert to** which converts an element to a new element of a different type

**Replace with** which replaces one element with another based on the user's choice

Rational Rhapsody [24] offers one simple refactoring, *Rename element* which can rename any element.

Poseidon UML [21] had Boger's et al. [4] refactoring browser implemented as a plugin. It is currently not included in the software unless specifically installed as a plugin.

Some academic tools have also been provided. Philipp Seuring [40] created the GaliciaUML framework for using the Multi FCA algorithm for refactoring UML. The framework is based on Eclipse and the EMF. However, the work is not focused on the refactorings themselves, only the framework behind running the transformations.

Pieter Van Gorp et al. [22] add a few minimal extensions to the UML metamodel to aid in keeping consistency between UML and code when refactoring. Pre- and postconditions are composed in OCL to verify preservation of program behaviour. They focus on the UML model and not on refactoring UML diagrams.

### 3.3. Discussion

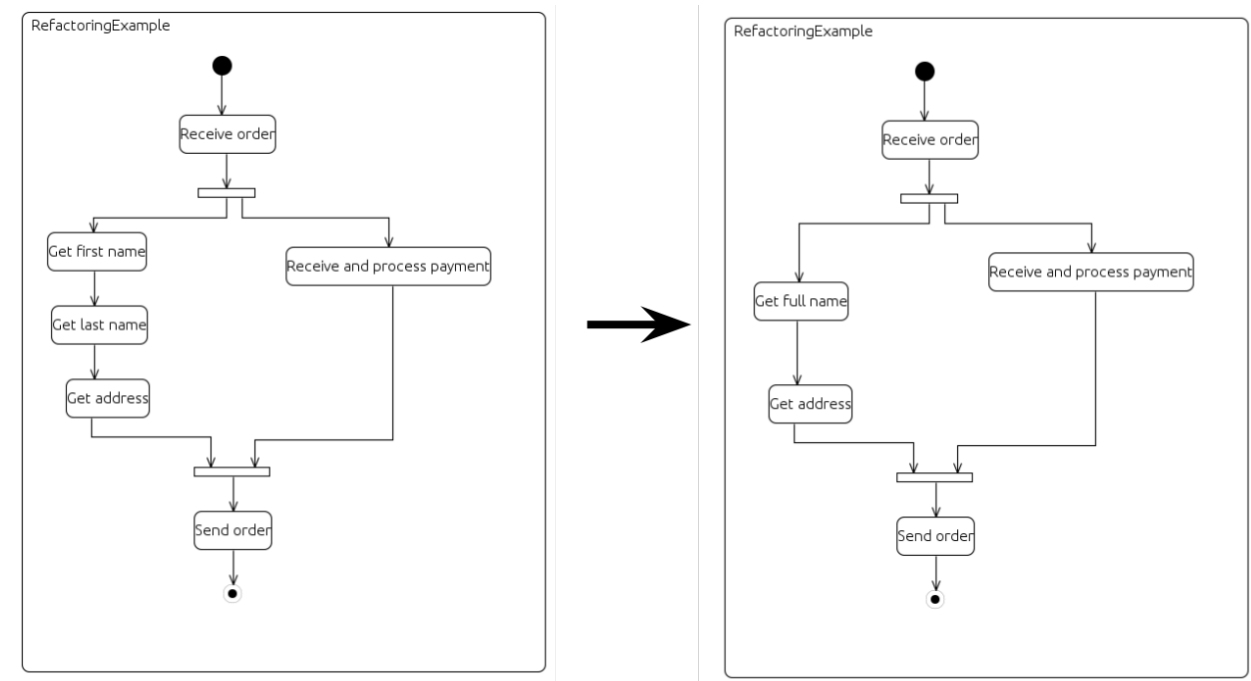
As this chapter shows, refactoring UML models and diagrams is quite an active field of research, but there is still work to do. Little work has been done on UML diagrams other than class diagrams and state machine diagrams and almost no research has been done on refactoring the graphical models in parallel to the UML model, which is the topic of this thesis.

## 4. UML Refactorings

In this chapter, example refactorings developed for UML activity diagrams are described. The refactorings will be introduced in an abstract way from the user's point of view, but in chapter 5 they will be explained in more detail. There are two refactorings that are developed in this thesis: Merge actions and Divide action.

### 4.1. Merge actions

Two actions are merged into one, as demonstrated in Figure 4.1.



*Figure 4.1: The Merge actions refactoring*

### 4.1.1. Motivation

When working with activity diagrams, it can sometimes be the case that two actions are doing more or less the same thing. An example might be an action that receives the last name of a client and then right after that there is an action that receives the first name of a client. If there is no need for keeping the first and last names separate they can be merged into one action which receives the full name of the client, as demonstrated in Figure 4.1. This is beneficial to the user because it can be prone to errors for the user to do this manually as she will have to move all edges from an action to the other and then remove one of them.

### 4.1.2. Refactoring mechanics

This refactoring will merge two actions, which are linked together by an edge, into one. The action which is on the source end of the connecting edge will be merged into the original action and subsequently removed. There is one precondition that must be satisfied for the refactoring to be executed, the actions to be merged must be linked together by a control flow edge.

The steps in the refactoring are:

- Move the source of all outgoing edges from the action to be removed to the action that will be kept
- Move the target of all incoming edges from the action to be removed to the action that will be kept
- Add name of action to be removed to the merged action
- Remove the control flow edge that connects the actions to be merged
- Remove the source action

## 4.2. Divide action

An action is divided into two, as demonstrated in Figure 4.2.

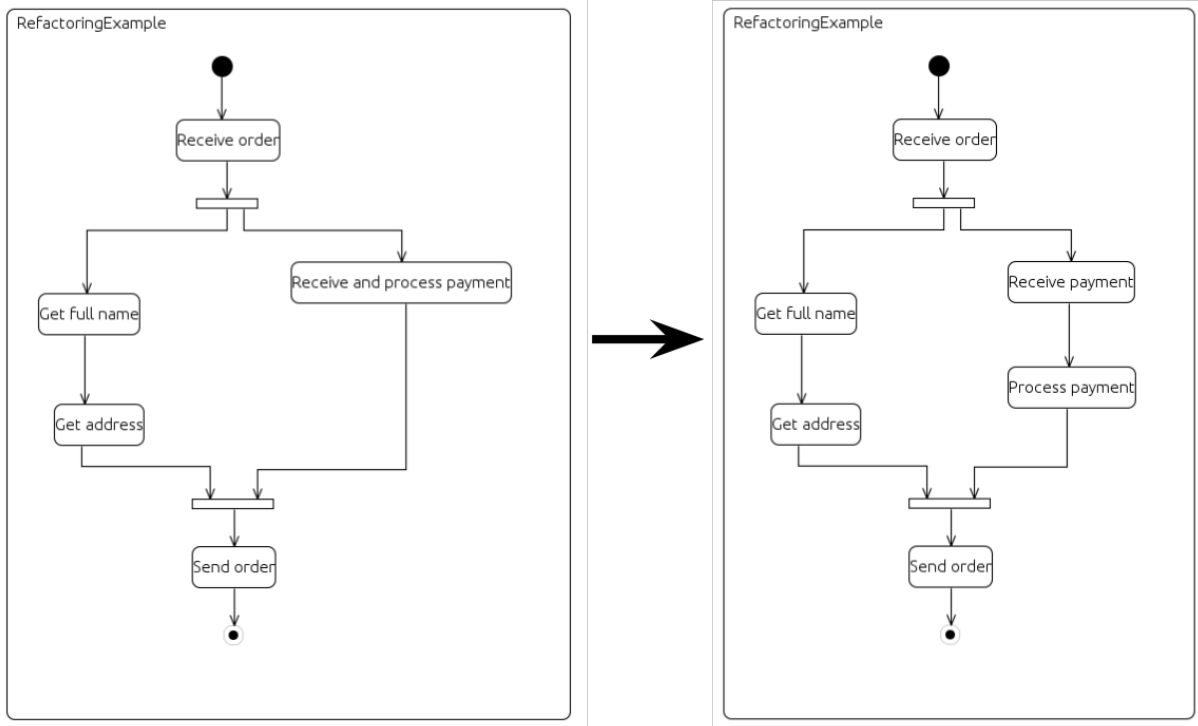


Figure 4.2: The Divide actions refactoring

### 4.2.1. Motivation

Sometimes an action is doing more than it should be doing, possible symptoms may be long names, or names that contain the word "and". An example might be an action that receives a payment and processes it. It would be more clear to divide the action into two actions, where the first one receives a payment and another that processes it, as Figure 4.2 demonstrates. It is beneficial for the user to be able to do this automatically because it takes time to create a new action, position it and then move all the outgoing edges to the new action.

### 4.2.2. Refactoring mechanics

This refactoring will divide one action into two by making a copy of the original and adding a control flow edge between them.

The steps in the refactoring are:

#### 4. *UML Refactorings*

- Make a clone of the original action
- Set coordinates of new action below original action
- Set name of new action to "New [name of original action]"
- Add a control flow edge from the original to the clone
- Move the source of all outgoing edges from the original to the clone

## 5. Tool Implementation

In this chapter the implementation of the automated refactorings described in the previous chapter will be explained in detail. The reasons for choosing model-to-model transformation are discussed in section 5.1 and the tools used are described in section 5.2. The technological challenges are listed in section 5.3. Section 5.4 presents the key classes in the UML metamodel and the Notation metamodel. The plugin architecture is outlined in section 5.5. Each of the most crucial components are then explained in detail in sections 5.6 and 5.7. The *Merge Actions* refactoring implementation is explained in section 5.8 and the Divide actions refactoring implementation is detailed in section 5.9.

### 5.1. Model-based vs. low-level transformations

Model-to-model transformations using transformation languages is a recent and proven way of refactoring models, as described in the previous chapter. A low-level solution for implementing the refactorings could be written with a conventional programming language but there are a few downsides. It would of course be quite a big task and would probably be closely tied to the XMI representation of both the UML model and the diagram, so the refactoring implementations might not be easily reused for other representations of a UML model and diagram. As model-to-model transformations can be run on any model, provided that a metamodel has been specified, the refactorings can probably be converted to run for other types of UML models and diagrams so using them instead of a low-level solution is a logical choice.

### 5.2. Tools

There are a couple of tools used in the development of the refactoring implementations:

- Eclipse IDE 3.6 (Helios)

## 5. Tool Implementation

- Java 1.6
- Eclipse Modeling Framework 2.6.1
- UML2 3.1.2
- Operational QVT for Eclipse 3.0.1
- Papyrus UML 0.7.0

The main reasons for choosing the Eclipse platform for implementing these refactorings is that its components are open source and that the EMF is feature rich and based on OMG standards. QVTO is also based on an OMG standard like the UML and some work has already been done on refactorings before with QVTO which makes it a good candidate for implementing the refactorings, even though it is not as mature as ATL. ATL is not based on an official OMG standard, so using standardized QVTO enables reuse of the refactoring transformations in other tool contexts.

There exist at least two open source UML diagram editors for Eclipse, UML2tools [13] and Papyrus UML. UML2Tools is not supported in the current version of Eclipse and continued support is not planned so Papyrus UML is the best choice for this, it is quite feature rich, supports all popular UML diagrams and is actively maintained.

### 5.3. Challenges

There are a few technological challenges in developing refactorings for UML models and the diagrams using QVTO. They are discussed in the following.

#### 5.3.1. Documentation

One of the main problems is the lack of documentation available for QVTO. The main reference document is the OMG's standard but as QVTO is still work in progress a part of the functionality is missing, some of which are detailed in this section.



### 5.3.2. Finding selected objects in the model

Another challenge is knowing which objects the user selected in Papyrus to refactor. The only way to be sure that the right object is selected is to reference the globally unique identifier (GUID) described in section 2.3.3. However, QVTO does not yet support the `_globalId()` method which is described in the OMG's standard for QVT [26] and intended for getting the GUID of an object. The metamodels do not provide any means for accessing the GUID either, so another method has to be used to solve this problem. The solution is demonstrated in section 5.6.1.

### 5.3.3. Creating a new object in the model

Creating a new object in a model can be a difficult task. Most model objects have many properties and subobjects that can not be explicitly accessed or set through the metamodel. There exist some options to solve this in QVTO, like the `deepclone()` operation which copies an element and all its subobjects to make a new one. However, it does not create a GUID for the new element until the next time the model is saved by the user, so that problem has to be solved as well. The solution is outlined in section 6.3.

### 5.3.4. Accessing coordinates of diagram elements

The metamodel for the Papyrus Notation diagram does not provide a way of accessing the coordinates of an element, or changing them. QVTO does not provide this functionality either, so this has to be achieved in another way. The solution is shown in section 5.6.2.

### 5.3.5. Transforming two models

For refactoring on the model level and on the diagram level at the same time, the diagram notation model and the UML model and the references between them have to be respected. QVTO supports transforming more than one model in one transformation very well, but the main challenge is that the same object is referenced in two different models, even though one refers to it as a diagram element and the other as a model element. Fortunately the Notation metamodel for Papyrus diagrams includes a reference to the UML model for each model object referenced in a diagram, but the UML metamodel does not include a reference to the diagram

element that corresponds to a model object. The solution is outlined in sections 6.2 and 6.3.

### 5.4. Metamodel classes

The Notation and UML metamodels have many different classes and complex relationships between them. However, to understand the transformations presented in this chapter it is sufficient to know only a small set of them.

#### 5.4.1. Notation metamodel classes

For the proprietary Notation metamodel used by Papyrus, the following classes are of importance:

**Diagram** The *Diagram* class is a top level class in the notation that contains all other elements as children.

**Shape** The *Shape* class extends the *Node* class, it represents every node in a diagram that has a shape. A *Shape* has a few properties of note:

**sourceEdges** The *sourceEdges* property is a set which contains all outgoing edges from the *Shape* object.

**targetEdges** The *targetEdges* property is a set which contains all incoming edges to the *Shape* object.

**element** The *element* property holds a reference to the UML model element the *Shape* object represents.

**LayoutConstraint** The *LayoutConstraint* class is owned by a *Shape* object and holds information about the layout of the *Shape*, i.e. position and size.

**Bounds** The *Bounds* class is descended from the *LayoutConstraint* class and represents the position of a *Shape* element.

**DecorationNode** A *DecorationNode* is a node that has no shape. The model for an activity diagram contains a *DecorationNode* which in turn contains all diagram nodes as children, and can be accessed with the *children* property.

**Edge** The *Edge* class represents every edge in a diagram. The edges belong to the *Diagram* class and can be accessed with the *edges* property of the *Diagram* class.

### 5.4.2. UML metamodel classes

The UML metamodel is very complex, but for this implementation only these classes are important:

**Activity** The *Activity* class is a top level class which contains all elements belonging to an activity diagram. It has a few important properties:

**ownedElement** The *ownedElement* property is a set which contains all elements of the activity diagram.

**node** The *node* property is a set which contains all nodes in the activity diagram.

**edge** The *edge* property is a set which contains all edges in the activity diagram.

**ActivityNode** The *ActivityNode* class is an abstract class that represents every node in an activity diagram, including actions and decision nodes.

**OpaqueAction** The *OpaqueAction* class is an implementation of the *ActivityNode* class and represents an ordinary action which is an important part of an activity diagram.

**ActivityEdge** The *ActivityEdge* class is an abstract class that represents every edge in an activity diagram, including control flows and links.

**ControlFlow** The *ControlFlow* class is an implementation of the *ActivityEdge* class and represents the control flow edge which defines the flow between actions.

## 5.5. Plugin architecture

The implementation of the refactorings is built as a plugin to the Eclipse IDE. It depends on QVTO and Papyrus UML and can be easily extended. This plugin has three components:

## 5. Tool Implementation

**is.hi.cs.umlrefactoring.core.libraries** In this component the Java blackboxing library is contained and made available to other plugins.

**is.hi.cs.umlrefactoring.ui** In this component all the Java code is contained to invoke the transformations. It also contains definitions for adding context menu entries to the Papyrus editor.

**is.hi.cs.umlrefactoring.core.transformations** This component contains the QVTO transformations.

The structure of the plugin is shown in Figure 5.1.

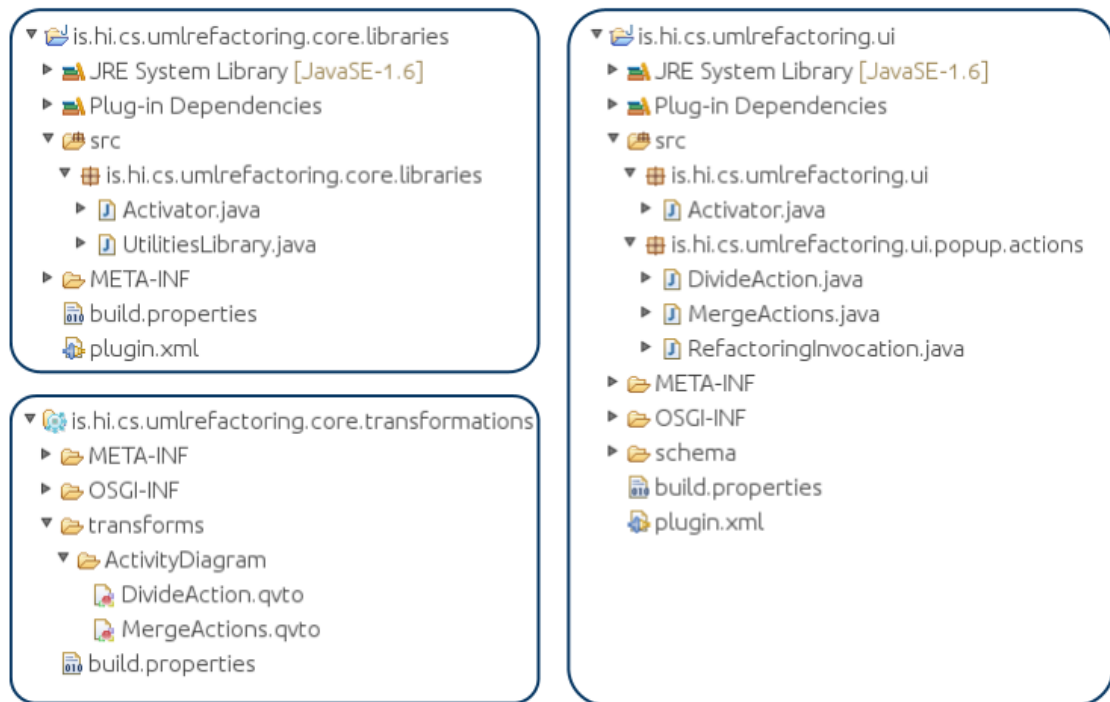


Figure 5.1: The structure of the plugin

The *is.hi.cs.umlrefactoring.core.transformations* component includes two QVTO transformation files, *MergeActions.qvto* and *DivideActions.qvto* in the *ActivityDiagram* directory. It also contains automatically generated plugin description files created by Eclipse.

The *is.hi.cs.umlrefactoring.ui* component includes three files in the *is.hi.cs.umlrefactoring.ui.popup.actions* package which take care of the QVTO invocation:

**RefactoringInvocation.java** *RefactoringInvocation* is an abstract class that con-

tains a *run()* method, described in section 5.7.1, which the *MergeActions* and *DivideAction* classes inherit as subclasses. It contains generic re-usable code for invoking the refactoring implementations. It also defines three abstract methods which are implemented in the subclasses.

**MergeActions.java** The *MergeActions* class handles the *Merge actions* refactoring and implements the abstract methods defined in the *RefactoringInvocation* superclass.

**DivideAction.java** The *DivideActions* class handles the *Divide actions* refactoring and implements the abstract methods defined in the *RefactoringInvocation* superclass.

The component also contains a *plugin.xml* file which adds context menu entries to the Papyrus editor. It is described in detail in section 5.7.2.

The *is.hi.cs.umlrefactoring.libraries* component includes the *UtilitiesLibrary.java* file which contains the blackboxing library detailed in section 5.6. The component also contains a *plugin.xml* file which makes the blackboxing library available to other components. It is described in detail in section 5.6.

### 5.5.1. Workflow

The workflow of the refactoring transformation is shown in Figure 5.2.

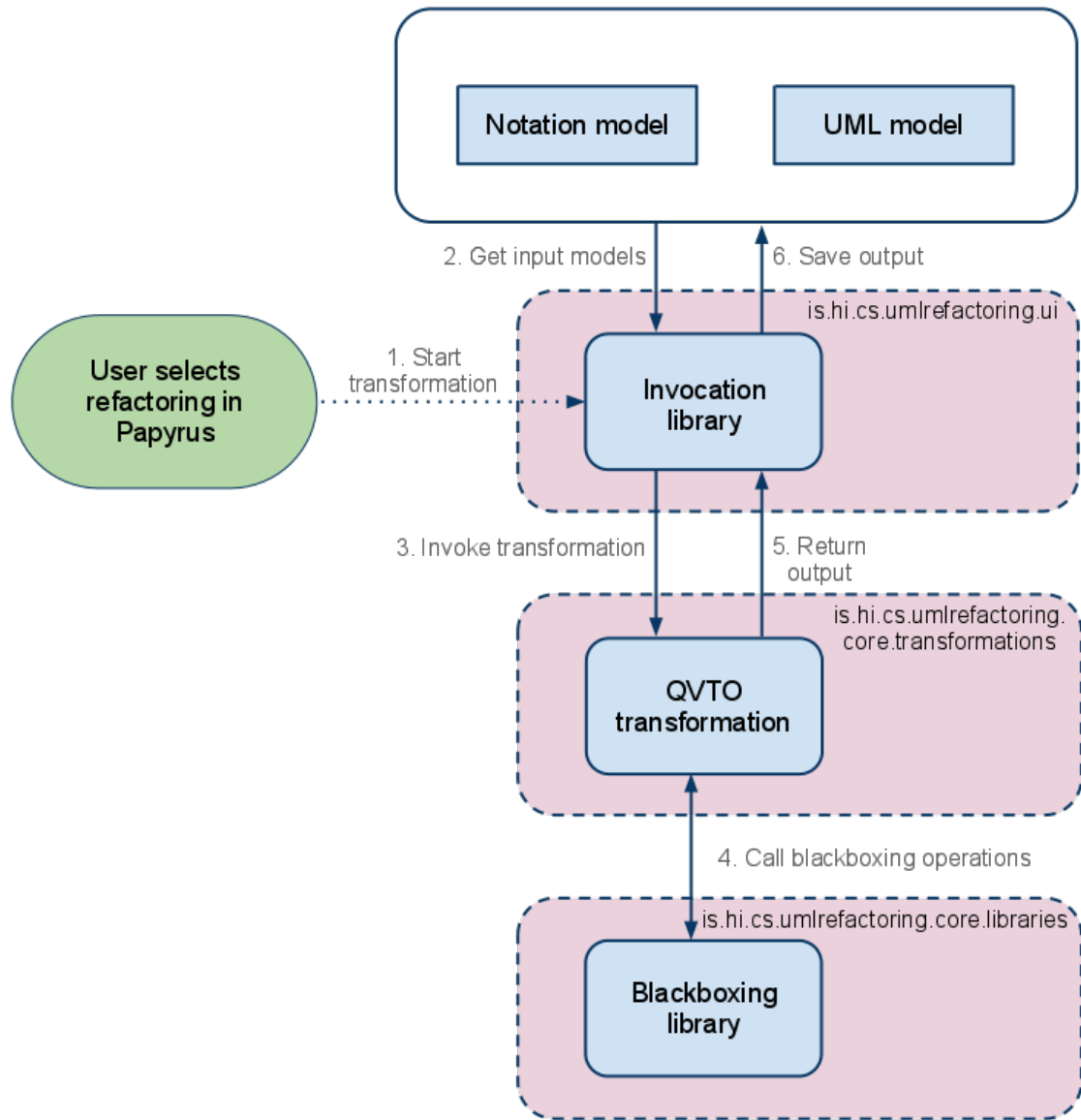


Figure 5.2: The workflow of the refactoring transformation

As seen in Figure 5.2, when the user selects a refactoring in the Papyrus UML editor, the invocation library starts with fetching the model files. It then sends the models to the QVTO transformation which calls blackboxing operations when needed. When the transformation has been executed, the invocation library saves the output, overwriting the original models.

## 5.6. Blackboxing library

As some of the problems listed in section 5.3 can not be solved in QVTO, other solutions are needed. The blackboxing method described in chapter 2 is a good way of solving these kinds of problems. With that approach, Java can be utilized to access and change properties which can not be accessed through the metamodel in QVTO.

This can be achieved by creating a Java class with the operations needed and importing it to the QVTO transformation. To make it available to QVTO the plugin.xml file has to include the code shown in Listing 5.1. The listings will generally only show the code that is relevant to each part so often the line numbers do not start with the first line.

```

4 <extension point="org.eclipse.m2m.qvt.oml.javaBlackboxUnits">
5   <unit name="UmlUtilities" namespace="m2m.qvt.oml">
6     <library name="UtilitiesLibrary" class="is.hi.cs.umlrefactoring.
7       ui.UtilitiesLibrary">
8       <metamodel nsURI="http://www.eclipse.org/emf/2002/Ecore"/>
9     </library>
10  </unit>
11 </extension>

```

*Listing 5.1: Making a blackboxing library available in plugin.xml*

This XML code simply makes the Java *UtilitiesLibrary* class in line 6 located in the library *is.hi.cs.umlrefactoring.ui.UtilitiesLibrary* available to QVTO as *org.eclipse.m2m.qvt.oml.javaBlackboxUnits* in line 4.

For implementing the refactorings described in the previous chapter, four operations are needed in the *UtilitiesLibrary* class:

- `isSelectedObject()`
- `moveElement()`
- `setElementID()`
- `generateElementID()`

The operations are explained in detail below, excluding the *generateElementID()* operation which is a very simple helper for the *setElementID()* operation.

### 5.6.1. isSelectedObject()

This is an important operation for solving the problem of reliably finding the selected object in the model, i.e. to which object or objects a refactoring shall be applied. The code is shown in Listing 5.2:

```

19 public static boolean isSelectedObject(Object shapeElement, String
    fragment) {
20     String elementFragment = ((EObject) shapeElement).eResource().
        getURIFragment((EObject) shapeElement);
21     return elementFragment.equals(fragment);
22 }

```

*Listing 5.2: The isSelectedObject() blackboxing operation*

The boolean operation in Listing 5.2 gets two parameters as input; *shapeElement* which is a Shape object and *fragment*, a string which contains the GUID of an element. In line 20 the GUID of the element is fetched and in line 21 it is compared to the input GUID to check if they match. If they match, that means that a selected element has been found and *true* is returned to the calling transformation.

### 5.6.2. moveElement()

The *moveElement()* operation in Listing 5.3 receives a Bounds element which contains coordinates and x and y offsets to move the element from its original position. It then returns the element. The operation is used in this implementation for moving a copied diagram element from its original position to a new position in the diagram.

```

25 public static Object moveElement(Object boundsElement, int x, int y) {
26     Bounds element = ((Bounds) boundsElement);
27     element.setX(element.getX() + x);
28     element.setY(element.getY() + y);
29     return element;
30 }

```

*Listing 5.3: The moveElement blackboxing operation*

The code in Listing 5.3 is quite simple, in lines 27 and 28 the element is moved x pixels horizontally and y pixels vertically, before being returned back to the caller in line 29.



### 5.6.3. setElementID()

The `setElementID()` operation shown in Listing 5.4 gets two parameters as input; *shapeElement* and *oldElement* which are both Shape objects. The GUID is generated for the *shapeElement* object, but another object is needed to be passed as a parameter to the function so the underlying resource can be accessed. As the *shapeElement* is created in QVTO before this function is called, it doesn't contain the Ecore resource which in turn contains all the other elements in the model. We need to be able to access the Ecore resource to be sure that there is no other object in the model that has the same GUID.

```

33 public staticEObject setElementID(Object shapeElement, Object
    oldElement) {
34    EObject oldElem = (EObject) oldElement;
35     String idRandomPart;
36
37     // Create a new ID for the element and
38     // make sure that an element with this ID doesn't exist
39     do {
40         idRandomPart = generateElementID();
41     } while (oldElem.eResource().getEObject(idRandomPart) != null);
42
43     XMIResource resource = (XMIResource) oldElem.eResource();
44     resource.setID((EObject) shapeElement, "_SpltEl" + idRandomPart.
        toString());
45     return ((EObject) shapeElement);
46 }

```

Listing 5.4: The `setElementID()` blackboxing operation

This function simply starts with casting the *oldElement* object to an Ecore object. In line 40 the GUID is generated using the *generateElementID()* function which is a very simple function that randomly picks characters from the alphabet to generate the GUID. The do-while loop then makes sure that the ID is truly unique in the model. In lines 43-45 the object's GUID is set and then returned to the caller.

## 5.7. Executing the transformation

The QVTO plugin for Eclipse provides the `TransformationExecutor` class for invoking QVTO transformations in Java as described in section 5.7.1. Such an invocation is typically initiated via the user interface. The extension point functionality provided by Eclipse can then be used to add context menu entries to the Papyrus editor so the refactorings can be easily selected in the user interface. This is described in detail in section 5.7.2.

### 5.7.1. Invoking a QVTO transformation with Java

The code in this implementation is based on the example of using the `TransformationExecutor` class provided by the QVT Eclipse project [15], but with a few modifications. The `run()` method in Listing 5.5 is located in the abstract `RefactoringInvocation` class, and the `DivideAction` and `MergeAction` classes inherit it. This design is based on the template method design pattern from Gamma et al. [20]. Using this design pattern means that adding a transformation is an easy task because the `run()` method can be reused and only three methods have to be implemented in the subclass for the transformation, `getSelection()`, `initTransformation()` and `setProperties()`, which are explained later in the section. Only the methods in the `MergeActions` class will be explained as they are very similar to the ones in the `DivideActions` class.

#### The `run()` method

The `run()` method in Listing 5.5 runs the transformation and saves the output, and handles possible errors that can occur during transformation. The details are described below.

```

48 public void run(IAction action) {
49     List<ShapeImpl> selectedItems = getSelection();
50
51     // Get path to diagram
52     IWorkbenchPage activePage = PlatformUI.getWorkbench().
        getActiveWorkbenchWindow().getActivePage();
53     CoreMultiDiagramEditor activeEditor = (CoreMultiDiagramEditor)
        activePage.getActiveEditor();
54     EObject model = (EObject) activeEditor.getDiagramEditPart().getModel
        ();
55     URI uri = model.eResource().getURI();
56     Path pathToDiagram = new Path(uri.toPlatformString(false));
57
58     if(!activeEditor.isDirty()) {
59         TransformationExecutor executor = initTransformation();
60
61         // define the transformation input
62         // Remark: we take the objects from a resource, however
63         // a list of arbitrary in-memory EObjects may be passed
64         ResourceSetImpl resourceSet = new ResourceSetImpl();
65         Resource umlInResource = resourceSet.getResource(URI.createURI("
            platform:/resource" + pathToDiagram.removeFileExtension().
            addFileExtension("uml").toString()), true);
66         EList<EObject> umlInObjects = umlInResource.getContents();
67
68         Resource notationInResource = resourceSet.getResource(URI.
            createURI("platform:/resource" + pathToDiagram.toString()),

```

```

69         true);
70     EList<EObject> notationInObjects = notationInResource.getContents
71         ();
72     // create the input extent with its initial contents
73     ModelExtent umlInput = new BasicModelExtent(umlInObjects);
74     ModelExtent notationInput = new BasicModelExtent(
75         notationInObjects);
76     // create an empty extent to catch the output
77     ModelExtent output = new BasicModelExtent();
78
79     ExecutionContextImpl context = setProperties(selectedItems);
80
81     // run the transformation assigned to the executor with the given
82     // input and output and execution context -> ChangeTheWorld(in,
83     // out)
84     // Remark: variable arguments count is supported
85     ExecutionDiagnostic result = executor.execute(context,
86         notationInput, umlInput);
87
88     // check the result for success
89     if(result.getSeverity() == Diagnostic.OK) {
90         // the output objects got captured in the output extent
91         List<EObject> outObjects = output.getContents();
92         // let's persist them using a resource
93         Resource umlOutResource = umlInResource;
94         Resource notationOutResource = notationInResource;
95         umlOutResource.getContents().addAll(outObjects);
96         notationOutResource.getContents().addAll(outObjects);
97         try {
98             umlOutResource.save(Collections.emptyMap());
99             notationOutResource.save(Collections.emptyMap());
100         } catch (IOException e) {
101             e.printStackTrace();
102         }
103     } else {
104         // turn the result diagnostic into status and send it to error
105         // log
106         IStatus status = BasicDiagnostic.toIStatus(result);
107         Activator.getDefault().getLog().log(status);
108     }
109 }
110 }

```

Listing 5.5: The run() method in RefactoringInvocation.java

## 5. Tool Implementation

The *run()* method in Listing 5.5 starts with running the *getSelection()* method, which is explained in section 5.7.1, to populate the *selectedItems* list with the selected objects. In lines 52-56 the path to the diagram that contains the selected items is looked up and stored in the *pathToDiagram* variable.

Line 58 checks if any unsaved changes exist in the editor by looking at the *isDirty()* property of the editor. If unsaved changes exist then an error message is displayed to the user in lines 104-109. Running a transformation on a model when changes have not been saved to it will cause the changes to be lost, because the user might have added a new element to the diagram but the element will not appear in the underlying models until the diagram is saved in Papyrus.

If no unsaved changes exist, the preparation for the execution continues in line 59 by running the *initTransformation()* method which is explained in section 5.7.1, which returns an *executor* object of the *TransformationExecutor* class.

In lines 64-75 the underlying UML and Notation models are looked up and loaded into the *umlInObjects* and *notationInObjects* variables respectively. It should be noted that the *pathToDiagram* variable holds the path to the .di file of the Notation model, but as the Notation and UML model files have the same name as the .di file, except the file extension, it is simply enough to replace the extension of the file with the correct one as is done in lines 65 and 68 with the *removeFileExtension()* and *addFileExtension()* methods. In line 75 an empty extent is created to capture the output from the transformation.

The *setProperties()* method is called in line 77 to create an execution context and pass properties to the transformation. This is explained in detail in section 5.7.1. The transformation is finally executed in line 82 by calling the *execute()* method on the *executor* object with the execution context and model objects as parameters. The result of the transformation is kept in the result variable which holds information about whether the transformation was executed and any errors that might have occurred.

The *result* object is checked in line 85 to see if the transformation was executed correctly. If errors occurred, the execution flow goes to lines 99-103 where the error descriptions are extracted from the *result* object and sent to an error console in the Eclipse IDE. However, if the transformation executed without errors the execution continues in lines 87-98. The transformed models are then gathered from the *output* model extent and loaded into the *outObjects* list in line 87. In lines 89-92 the resource of the input models is replaced with the output models, however, no data is written before the resources are saved. In lines 94 and 95 the UML and Notation model resources are saved, essentially overwriting the input models with the transformation output. The Papyrus tool will then update the diagram display.

### The `getSelection()` method

The *getSelection()* method returns a list of the items the user selected as *ShapeImpl* objects. The implementation in the *MergeActions* class is shown in Listing 5.6

```

40 public List<ShapeImpl> getSelection() {
41     List<ShapeImpl> selectionList = new ArrayList<ShapeImpl>();
42     Object[] selections = selection.toArray();
43
44     // Get selected items
45     OpaqueActionEditPart firstEditPart = (OpaqueActionEditPart)
        selections[0];
46     OpaqueActionEditPart secondEditPart = (OpaqueActionEditPart)
        selections[1];
47     ShapeImpl firstItem = (ShapeImpl) firstEditPart.getModel();
48     ShapeImpl secondItem = (ShapeImpl) secondEditPart.getModel();
49     selectionList.add(firstItem);
50     selectionList.add(secondItem);
51     return selectionList;
52 }

```

Listing 5.6: The *getSelection()* method in *MergeActions.java*

The method starts with creating an empty list intended for *ShapeImpl* objects in line 41 and creates an array of the selected objects in line 42. In this implementation two and only two items are expected to be in the *selection* array because the refactoring is implemented for merging two actions. In lines 45-51 the first two items are fetched from the array and added to the *selectionList* as *ShapeImpl* objects. The list is then returned back to the caller.

### The `initTransformation()` method

The *initTransformation()* method returns an object of the *TransformationExecutor* class which includes the path to the resource containing the QVTO transformation. The implementation in the *MergeActions* class is shown in Listing 5.7

```

54 public TransformationExecutor initTransformation() {
55     URI transformationURI = URI.createURI("platform:/plugin/is.hi.cs.
        umlrefactoring.core.transformations/transforms/ActivityDiagram/
        MergeActions.qvto");
56     TransformationExecutor executor = new TransformationExecutor(
        transformationURI);
57     return executor;
58 }

```

Listing 5.7: The *initTransformation()* method in *MergeActions.java*

## 5. Tool Implementation

The code in Listing 5.7 starts with creating a URI containing the path to the QVTO transformation contained within the plugin in line 55. The object of the *TransformationExecutor* class is created with the URI to the transformation in line 56 and then returned to the caller in line 57.

### The `setProperties()` method

The *setProperties()* method sets the configuration properties for the transformation with the *selectionList* parameter which contains the selected items from the *getSelection()* method. It returns an object of the *ExecutionContextImpl* class to the caller. The implementation in the *MergeActions* class is shown in Listing 5.8

```
60 public ExecutionContextImpl setProperties(List<ShapeImpl> selectionList
    ) {
61     ExecutionContextImpl context = new ExecutionContextImpl();
62
63     context.setConfigProperty("toMerge1", selectionList.get(0).eResource
        ().getURIFragment(selectionList.get(0)));
64     context.setConfigProperty("toMerge2", selectionList.get(1).eResource
        ().getURIFragment(selectionList.get(1)));
65     return context;
66 }
```

*Listing 5.8: The `setProperties()` method in `MergeActions.java`*

The method in Listing 5.8 starts with creating an object of the *ExecutionContextImpl* class called *context*. The *setConfigProperty()* method is used in lines 63 and 64 to add the GUID strings of the selected elements, i.e. the activity node to be merged, as configuration properties to the QVTO transformation. The method then ends in line 65 with returning the *context* object to the caller.

### 5.7.2. Adding a context menu entry

In this plugin the *org.eclipse.ui.popupMenus* extension point is utilized to add an entry to the *OpaqueAction* element's context menu. This is specified in the *plugin.xml* file in the *is.hi.cs.umlrefactoring.ui* plugin. The code is shown in Listing 5.9.

```

4 <extension point="org.eclipse.ui.popupMenus">
5   <objectContribution
6     id="is.hi.cs.umlrefactoring.ui.popup.actions"
7     objectClass="org.eclipse.papyrus.diagram.activity.edit.parts.
      OpaqueActionEditPart"
8     nameFilter="*">
9   <menu
10    id="org.eclipse.gmf.runtime.diagram.ui.DiagramEditorContextMenu.
      Refactor"
11    label="Refactor"
12    path="additions">
13    <separator name="group1"/>
14  </menu>
15  <action id="is.hi.cs.umlrefactoring.ui.popup.actions.MergeActions"
16    label="Merge actions"
17    menubarPath="org.eclipse.gmf.runtime.diagram.ui.
      DiagramEditorContextMenu.Refactor/group1"
18    definitionId="is.hi.cs.umlrefactoring.ui.popup.actions.
      MergeActions"
19    class="is.hi.cs.umlrefactoring.ui.popup.actions.MergeActions"
20    enablesFor="2">
21  </action>
22  <action id="is.hi.cs.umlrefactoring.ui.popup.actions.DivideAction"
23    label="Divide action"
24    menubarPath="org.eclipse.gmf.runtime.diagram.ui.
      DiagramEditorContextMenu.Refactor/group1"
25    definitionId="is.hi.cs.umlrefactoring.ui.popup.actions.
      DivideAction"
26    class="is.hi.cs.umlrefactoring.ui.popup.actions.DivideAction"
27    enablesFor="1">
28  </action>
29  </objectContribution>
30 </extension>

```

Listing 5.9: Adding a context menu entry in plugin.xml

There are some important parts in this specification. Line 4 defines the extension point used, *org.eclipse.ui.popupMenus*, which provides a way of adding menu entries to any context menu in Eclipse and line 6 defines the identifier for the extension. Line 8 defines the type of object the context menu entry is added to, in this case for an action, defined as an *OpaqueActionEditPart* object by Papyrus.

Lines 9-14 define a submenu called *Refactor* where the individual refactoring actions are added.

Lines 15-21 specify the *Merge Actions* menu entry. Line 16 defines the label, Merge actions, and line 17 defines which menu the action is added to, which in this case is the submenu defined in lines 9-14. Line 18 defines an identifier for the entry and line 19 states which class is executed when the entry is selected. In line 20 it is defined that the menu entry is only enabled when two objects are selected.

Lines 22-28 specify the *Divide Action* menu entry, in the same way as the *Merge Actions* entry. However, it must be noted that the menu entry is only enabled when one object is selected.

### 5.8. Merge actions refactoring

The *Merge actions* transformation carries out the refactoring by the same name detailed in chapter 4. This section describes the QVTO code behind it and explains it line by line.

```

1 import m2m.qvt.uml.UmlUtilities;
2
3 modeltype NOTATION uses 'http://www.eclipse.org/gmf/runtime/1.0.2/
  notation';
4 modeltype UML uses 'http://www.eclipse.org/uml2/3.0.0/UML';
5 modeltype ECORE uses "http://www.eclipse.org/emf/2002/Ecore";
6
7 transformation MergeActions(inout notation : NOTATION, inout uml : UML)
  ;
8
9 configuration property toMerge1 : String;
10 configuration property toMerge2 : String;
11
12 property objToMerge1 : notation::Shape = null;
13 property objToMerge2 : notation::Shape = null;
14 property edgeToRemove : uml::ActivityEdge = null;
15 property incomingEdgesToTransform : Set(ActivityEdge) = Set{ };
16 property outgoingEdgesToTransform : Set(ActivityEdge) = Set{ };
17 property nodeToRemove : uml::ActivityNode = null;
18 property targetNode : uml::ActivityNode = null;
19
20 main() {
21   notation.objectsOfType(Shape) -> getObjects();
22   notation.objectsOfType(Shape) -> map merge();
23   uml.objectsOfType(ActivityEdge) -> map setTarget();
24   uml.objectsOfType(ActivityEdge) -> map setSource();
25   uml.objectsOfType(ActivityNode) -> map changeName();
26   uml.objectsOfType(ActivityEdge) -> map removeEdge();
27   uml.objectsOfType(ActivityNode) -> map removeNode();
28 }

```

*Listing 5.10: The MergeActions.qvto transformation*

The first line imports the blackboxing library which contains transformation helpers. Lines 3-5 define the metamodels used for the transformation: Notation, UML and Ecore.



Line 7 is the transformation header, which defines the name and input and output metamodels; in this case the input is the same as the output, Notation and UML.

Lines 9 and 10 define configuration properties which hold values that are passed into the transformation from the caller which in this case is the Java execution class. The properties, *toMerge1* and *toMerge2* get passed strings which contain the GUID of the respective diagram elements which were selected for merging.

Lines 12-18 define properties, or global variables, which can be referenced anywhere in the transformation. In this case they are used to store references to the selected objects to be merged and the UML elements that are to be transformed alongside the diagram objects.

In lines 20-28 is the main function which is the entry point to the transformation. In the main function the mappings are invoked as well as queries when applicable. It starts off with fetching the diagram objects whose GUID's were passed into the transformation as strings by calling the *getObjects()* query, which is explained in section 5.8.1.

In line 22 the actual merging is started by invoking the *merge()* mapping for the Notation model, which is detailed in section 5.8.2. Note the syntax for invoking the mapping:

```
notation.objectsOfType(Shape) -> map merge();
```

This code sends all objects of the Shape type from the Notation model to the *merge()* mapping. The mapping then includes a guard for checking whether the object passed is supposed to be transformed.

Lines 23-27 invoke mappings for the UML model, based on which diagram elements were transformed in the *merge()* mapping. This is explained in more detail in sections 5.8.2 and 5.8.3.

### 5.8.1. The *getObjects()* query

A query is a function that does not update any models but iterates over a set of input objects like a mapping. The *getObjects()* query receives a Shape as input and is void, that is, it doesn't return a value. However, it populates the global variables *objToMerge1* and *objToMerge2* with the relevant diagram elements.

```

30 query Shape::getObjects() : Void
31 {
32     if (self.isSelectedObject(toMerge1)) then {
33         objToMerge1 := self;
34     } endif;
35
36     if (self.isSelectedObject(toMerge2)) then {
37         objToMerge2 := self;
38     } endif;
39 }

```

Listing 5.11: The *getObjects()* query in *MergeActions.qvto*

The *getObjects()* query uses the blackboxing method *isSelectedObject()* in line 32 and line 36 to determine if the element in question is one of those selected for merging.

### 5.8.2. The *merge()* mapping

The *merge()* mapping is the most important part of the transformation. It transforms the Notation model and stores references to elements to be transformed in the UML model.

```

41 mapping inout notation::Shape::merge()
42   when { self.isSelectedObject(toMerge1) }
43   {
44       var firstNode : Shape = null;
45       var lastNode : Shape = null;
46       var connectingEdge : Edge = null;
47
48       self.targetEdges->forEach(incoming) {
49           if (incoming.source = objToMerge2) then {
50               firstNode := objToMerge2;
51               lastNode := objToMerge1;
52               connectingEdge := incoming;
53           } endif;
54       };
55
56       self.sourceEdges->forEach(outgoing) {
57           if (outgoing.target = objToMerge2) then {
58               firstNode := objToMerge1;
59               lastNode := objToMerge2;
60               connectingEdge := outgoing;
61           } endif;
62       };
63
64       if not (firstNode = null and lastNode = null) then {
65           edgeToRemove := connectingEdge.element.oclAsType(ActivityEdge);
66           nodeToRemove := firstNode.element.oclAsType(ActivityNode);

```

```

67     targetNode := lastNode.element.oclAsType(ActivityNode);
68     firstNode.oclAsType(Shape).targetEdges->forEach(incomingEdge) {
69         incomingEdge.target := lastNode;
70         incomingEdgesToTransform += incomingEdge.element.oclAsType(
            ActivityEdge);
71     };
72     firstNode.oclAsType(Shape).sourceEdges->forEach(outgoingEdge) {
73         outgoingEdge.source := lastNode;
74         outgoingEdgesToTransform += outgoingEdge.element.oclAsType(
            ActivityEdge)
75     };
76
77     notation.removeElement(connectingEdge);
78     notation.removeElement(firstNode);
79 }endif;
80 }

```

Listing 5.12: The *merge()* mapping in *MergeActions.qvto*

Line 41 states that the mapping takes a *Shape* as an input and outputs a *Shape*. Line 42 defines a guard which checks that the input *Shape* is the first one of the two selected using the *isSelectedObject()* blackboxing operation. It is sufficient to check for only one item because the transformation finds the other element by looking at nodes connected to it by an edge.

Lines 44-46 define three variables, *firstNode* and *lastNode*, which will be used to reference the node that comes first in the control flow of the nodes to be merged and the one that is last, respectively. the *connectingEdge* variable references the edge between the nodes to be merged.

In lines 48-62 the transformation looks at all incoming edges(*targetEdges*) and outgoing edges(*sourceEdges*) connected to the *toMerge1* object using the *forEach* construct, which scans through every item of a set. If the *toMerge2* object is found on the other end of an edge then the variables *firstNode*, *lastNode* and *connectingEdge* are set to reference the appropriate elements. The *if* statement in line 64 checks if the variables have been set and continues the transformation if so. If the variables are not set, that means that the elements selected are not connected by an edge and then the transformation is not executed.

If the transformation is to be executed, the *edgeToRemove* global property is in line 65 set to reference the UML object that corresponds to the *connectingEdge* object for a later use. The *oclAsType()* function is used for converting objects into another type, in line 65 it is used to convert the Ecore element that the *connectingEdge* element references to a UML edge. In lines 66 and 67 references to the UML elements to be removed and the node to be kept are stored in global properties, *nodeToRemove* and *targetNode*, respectively.

## 5. Tool Implementation

In lines 68-71 every outgoing edge from the *firstNode* shape is transformed. Its target is in line 69 set to the *lastNode* shape, which means that every edge that was connected to the *firstNode* shape will be connected to the *lastNode* shape. A reference to the UML element of every edge is also stored in the global *incomingEdgesToTransform* set.

Because the *firstNode* object will be removed, it is also necessary to move all outgoing edges from it to the *lastNode* object. This is done in lines 72-75 with a *forEach* construct, where the source of every edge is set to the *lastNode* shape and a reference to the UML element is added to the *outgoingEdgesToTransform* set.

The transformation on the notation model then ends with removing the *firstNode* and *connectingEdge* objects from the model in lines 77 and 78.

### 5.8.3. UML model element mappings

At this stage in the transformation the Notation model has been refactored according to the refactoring mechanics in chapter 4. Now all that is left to do is to reflect the changes in the UML model. That is quite a simple task now because all the logic has been taken care of in the Notation transformation, and global variables are in place that contain the UML elements to transform. The code for the UML mappings is in Listing 5.13.

```
82 mapping inout uml::ActivityEdge::removeEdge()  
83   when { edgeToRemove=self }  
84   {  
85     uml.removeElement(self);  
86   }  
87  
88 mapping inout uml::ActivityEdge::setTarget()  
89   when { incomingEdgesToTransform->includes(self) }  
90   {  
91     self.target := targetNode;  
92   }  
93  
94 mapping inout uml::ActivityEdge::setSource()  
95   when { outgoingEdgesToTransform->includes(self) }  
96   {  
97     self.source := targetNode;  
98   }  
99  
100 mapping inout uml::ActivityNode::removeNode()  
101   when { self = nodeToRemove }  
102   {  
103     uml.removeElement(self);  
104   }
```

```

105
106 mapping inout uml::ActivityNode::changeName()
107     when { self = targetNode }
108     {
109         self.name := nodeToRemove.name + " - " + self.name;
110     }

```

*Listing 5.13: The UML mappings in MergeActions.qvto*

In lines 82-86 the edge between the elements to be merged is removed from the UML model. In lines 88-92 the source of all incoming edges from the element to be removed are set to the element to be kept. The same thing happens in lines 94-98 to the outgoing edges, except that the source is changed.

In lines 100-104 the *nodeToRemove* element is removed from the UML model and then the transformation ends in lines 106-110 with prepending the name of the merged element with the name of the removed element and a hyphen in between.

## 5.9. Divide action refactoring

The *Divide action* transformation carries out the refactoring by the same name detailed in chapter 4. This section describes the QVTO code behind it and explains it line by line.

```

1 import m2m.qvt.uml.UmlUtilities;
2
3 modeltype UML uses 'http://www.eclipse.org/uml2/3.0.0/UML';
4 modeltype NOTATION uses 'http://www.eclipse.org/gmf/runtime/1.0.2/
  notation';
5 modeltype ECORE uses "http://www.eclipse.org/emf/2002/Ecore";
6
7 transformation DivideAction(inout notation : NOTATION, inout uml : UML)
8     ;
9 configuration property toDivide : String;
10
11 property objectToDivide : notation::Shape = null;
12 property newEdge : notation::Edge = null;
13 property newShape : notation::Shape = null;
14
15 property umlEdgesToTransform : Set(ActivityEdge) = Set{ };
16 property newUmlNode : ActivityNode = null;
17
18 main() {
19     notation.objectsOfType(Shape) -> getObjectToDivide();
20     notation.objectsOfType(DecorationNode) -> map addNode();
21     notation.objectsOfType(Diagram) -> map addEdge();

```

## 5. Tool Implementation

```
22 |    uml.objectsOfType(Activity) -> map addNodeAndEdge();
23 |    uml.objectsOfType(ActivityEdge) -> map setSource();
24 | }
```

*Listing 5.14: The DivideActions.qvto transformation*

The *Divide actions* transformation starts off in a similar way as the *Merge actions* transformation. Line 1 imports the Java blackboxing library and lines 3-5 define the Notation, UML and Ecore metamodels. The transformation header in line 7 specifies that the transformation has two input models and two output models; Notation and UML.

Line 9 defines a configuration property that contains the GUID string of the Shape that is to be divided. Global properties to keep In lines 11-16 global properties are defined for keeping references to the selected object, Notation element to be added and UML elements that correspond to elements in the Notation model.

The main function starts with calling the getObjectToDivide() query in line 19. Then a new node is added to the Notation model with the *addNode()* mapping and an edge between the original and the new node with the *addEdge()* mapping. Lines 22 and 23 do the same for the UML model. The query and the mappings are explained in detail below.

### 5.9.1. The getObjectToDivide() query

The *getObjectToDivide()* query receives a Shape object as input and looks up the selected object to divide in the Notation model by iterating over a set of input objects.

```
26 | query Shape:: getObjectToDivide() : Void
27 | {
28 |     if (self.isSelectedObject(toDivide)) then {
29 |         objectToDivide := self;
30 |     } endif;
31 | }
```

*Listing 5.15: The getObjectToDivide() query in DivideActions.qvto*

The query in Listing 5.15 starts starts with using the *isSelectedObject()* operation in line 28 to find the object selected for division. When found, the global property *objectToDivide* is used to store a reference to the element in line 29.

### 5.9.2. The `addNode()` mapping

The *addNode()* mapping receives a `DecorationNode` object as input and adds a new node to the Notation model by making a copy of the original one.

```

33 mapping inout notation::DecorationNode::addNode()
34   when { self.children->includes(objectToDivide) }
35   {
36     var newNode:notation::Shape := objectToDivide.deepclone().
        setElementID(objectToDivide).oclAsType(Shape);
37     var newPos:notation::LayoutConstraint := newNode.layoutConstraint.
        moveElement(0, 100).oclAsType(LayoutConstraint);
38     newNode.layoutConstraint := newPos;
39
40     if (objectToDivide.sourceEdges->notEmpty()) then {
41       newEdge := objectToDivide.sourceEdges->first().deepclone().
        oclAsType(Edge);
42       newEdge.source := objectToDivide;
43       newEdge.target := newNode;
44     } else {
45       if (objectToDivide.targetEdges->notEmpty()) then {
46         newEdge := objectToDivide.targetEdges->first().deepclone().
        oclAsType(Edge);
47         newEdge.source := objectToDivide;
48         newEdge.target := newNode;
49       } endif;
50     } endif;
51
52     objectToDivide.sourceEdges->forEach(outgoingEdge) {
53       if (outgoingEdge <> newEdge) then {
54         outgoingEdge.source := newNode;
55         umlEdgesToTransform += outgoingEdge.element.oclAsType(
        ActivityEdge);
56       } endif;
57     };
58
59     newShape := newNode;
60     self.children += newNode;
61   }

```

Listing 5.16: The *addNode()* mapping in *DivideActions.qvto*

As can be seen in Listing 5.16, the *addNode()* mapping starts with a guard that checks whether the `DecorationNode` includes the object to divide in the *children* set in line 34. In line 36 a new node is created by cloning the original node with the *deepclone()* operation QVTO provides and a GUID is set with the *setElementID()* operation. If not set explicitly, the new model element will exist without a GUID until the next time the user saves the model. That would make it impossible to execute further refactorings on that element until the model is saved, so the safest way is to set it right away.

## 5. Tool Implementation

A new `LayoutConstraint` element is created in line 37 and moved 100 pixels down in the diagram by using the `moveElement()` blackboxing operation. The `LayoutConstraint` element is then set to belong to the new node in line 38. In a more sophisticated approach, an algorithm could be developed to find the right place in the diagram for the new element but in this approach the solution is kept simple because in most activity diagrams the flow goes straight down from the top. This could also be decided by the user via an input dialog before the refactoring is executed, but it would be hard to get completely right because a pixel is quite a small unit and often displayed differently on different monitors. The user anyway has to edit the name of the new action and it is not uncommon to request interaction from the user in an advanced code refactoring.

Lines 40-50 contain conditional statements that check if there exist any incoming or outgoing edges that connect to the element to be divided. When found, the first edge in either set is copied with the `deepclone()` operation and then connected to the original node and the cloned node by setting the `source` and `target` properties appropriately.

In lines 52-57 all outgoing edges from the original node are moved to the new node by setting the `source` property of each edge to point to the new element. The UML elements of each node are added to the global `umlEdgesToTransform` set so the same can be done for them later on in the transformation.

The mapping then ends with keeping a reference to the new node object in the global `newShape` property, and the node is added to the `children` set of the `DecorationNode` being mapped.

### 5.9.3. The `addEdge()` mapping

The `addEdge()` mapping is very simple, it adds the edge between the divided elements to the `edges` set of the `Diagram` element.

```
63 mapping inout notation::Diagram::addEdge()  
64   when { objectToDivide.diagram = self }  
65   {  
66     if not (newEdge = null) then {  
67       self.edges += newEdge;  
68     } endif  
69   }
```

Listing 5.17: The `addEdge()` mapping in `DivideActions.qvto`

The mapping in Listing 5.17 starts with a guard in line 64 that checks if the correct `Diagram` element is being mapped. When the correct `Diagram` element is found,



and the *newEdge* element exists, the edge is added to the *edges* set of the Diagram element in line 67.

#### 5.9.4. The addNodeAndEdge() mapping

The addNodeAndEdge() mapping receives a UML Activity element as input and adds an action and an edge to it to reflect the changes already done to the Notation model.

```

71 mapping inout uml::Activity::addNodeAndEdge()
72   when { self.ownedElement->includes(objectToDivide.element.oclAsType(
      uml::ActivityNode)) }
73   {
74     var newActivity : UML::ActivityNode := object uml::OpaqueAction {
75       name := "New " + objectToDivide.element.oclAsType(uml::
        ActivityNode).name;
76     };
77
78     if not (newEdge = null) then {
79       var newControlFlow : UML::ActivityEdge := object uml::ControlFlow
        {
80         name := "New Edge";
81         target := newActivity;
82         source := objectToDivide.element.oclAsType(uml::ActivityNode);
83       };
84       self.edge += newControlFlow;
85       newEdge.element := newControlFlow.oclAsType(ecore::EObject);
86     } endif;
87
88     newUmlNode := newActivity;
89     self.node += newActivity;
90     newShape.element := newActivity.oclAsType(ecore::EObject);
91   }

```

Listing 5.18: The addNodeAndEdge() mapping in DivideActions.qvto

The mapping in Listing 5.18 starts with a guard in line 72 that makes sure that correct activity element is mapped by checking that the original action is contained in the *ownedElement* set of the Activity.

When the correct element is found, an *OpaqueAction* node and a *ControlFlow* edge are created in lines 74-86. The *OpaqueAction* element is given the name of the original action with the text "New" prepended in line 75. The *ControlFlow* edge, if it exists, is given a name in line 80 and then its *source* and *target* properties are set to the original action and the new action in lines 81-82. In line 84 the edge is added to the *edge* set of the Diagram element. In line 85 the *element* property of

## 5. Tool Implementation

the corresponding edge from the notation model is set to reference the new UML control flow edge, which was not possible before the UML element had been created.

In line 88 a reference to the new action is stored in the global *newUmlNode* property and then the newly created node is added to the *node* set of the Diagram element in lines 89. The mapping then ends with setting the corresponding *element* property of the node from the Notation model.

### 5.9.5. The `setSource()` mapping

The last mapping of the transformation, the *setSource()* mapping, is quite simple. It receives an `ActivityEdge` as input and moves its source to the divided element.

```
93 mapping inout uml::ActivityEdge::setSource()  
94   when { umlEdgesToTransform->includes(self) }  
95   {  
96     self.source := newUmlNode;  
97   }
```

*Listing 5.19: The `setSource()` mapping in `DivideActions.qvto`*

The mapping in Listing 5.19 starts with a guard in line 94 that checks if the element to be mapped is contained in the *umlEdgesToTransform* set. In line 96 the *source* property is set to the divided node, *newUmlNode*, which concludes the transformation.

## 6. Evaluation

In this chapter the refactorings implemented in the plugin described in chapter 5 are evaluated. An activity diagram will be created with Papyrus in section 6.1 and used as an example to show that the refactorings work in sections 6.2 and 6.3. Finishing touches are added to the example in section 6.4 and the chapter is concluded with a discussion in section 6.5.

### 6.1. The example activity diagram

The activity diagram used as an example to show the refactorings is based on the activity diagram from Fowler shown in section 2.2.2 [19]. A papyrus project called RefactoringExample is created in Eclipse with the project creation wizard along with the example activity diagram contained in the project, with the steps shown in figures 6.1 to 6.6.

Creating the diagram only requires a few steps. A new Papyrus project is created by selecting the New menu entry in the file menu as demonstrated in Figure 6.1. A *Papyrus project* is selected in Figure 6.2 and given a name in Figure 6.3. The UML language is selected for the project in Figure 6.4 and the last step involves creating an activity diagram inside the project and naming it RefactoringExample as shown in Figure 6.5. Three files will be created inside the project; model.di, model.notation and model.uml. Note that the highlighted *Link with editor* button in Figure 6.6 has to be disabled before trying the refactorings because a bug in Papyrus will make the selection of more than one object impossible with the *Link with editor* option selected.

Finally the diagram is created by adding the relevant actions and control flows which will generate the diagram shown in Figure 6.7.

## 6. Evaluation

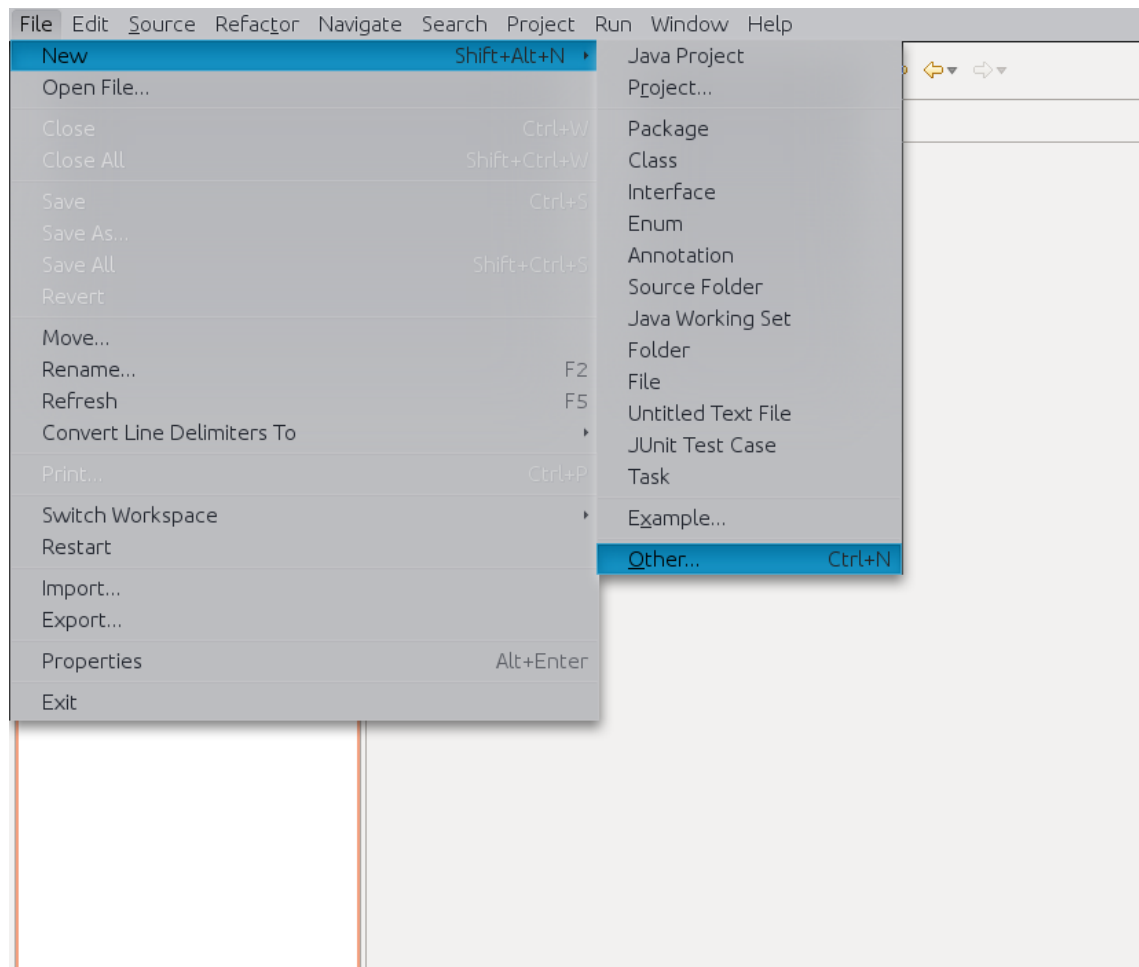


Figure 6.1: Creating a new project in Eclipse

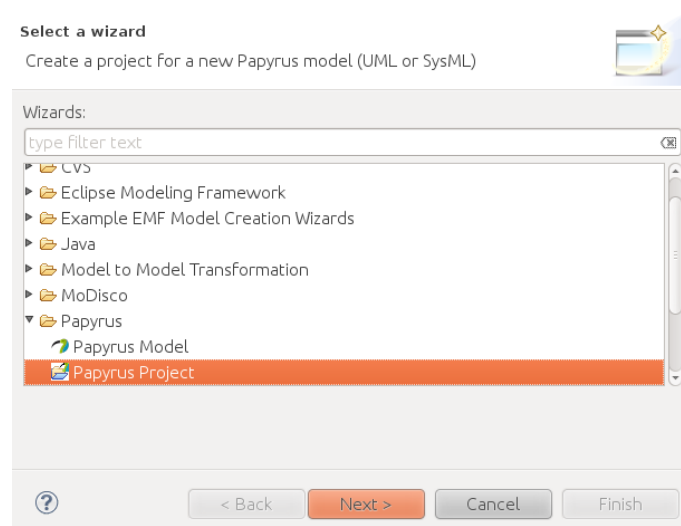
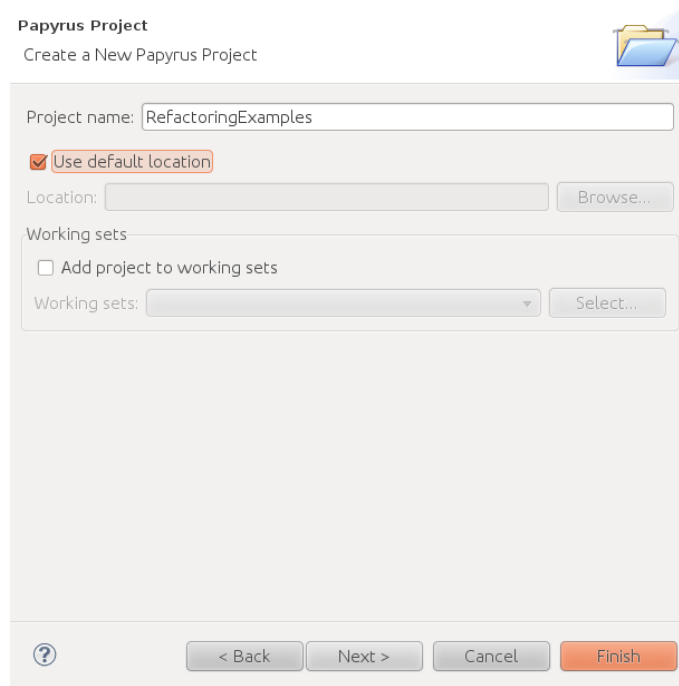


Figure 6.2: Creating a new Papyrus project in Eclipse

## 6.1. The example activity diagram



**Papyrus Project**  
Create a New Papyrus Project

Project name:

☒ Use default location

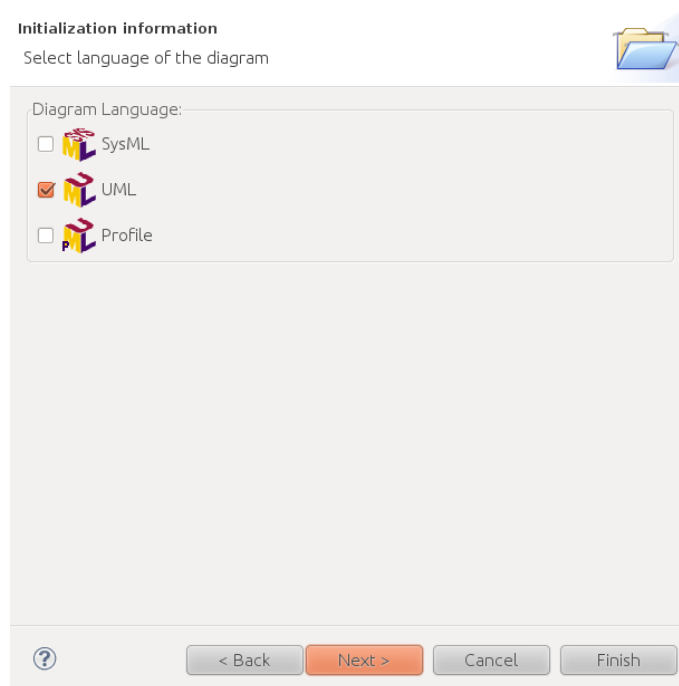
Location:

Working sets

☐ Add project to working sets

Working sets:

Figure 6.3: Naming the Papyrus project



**Initialization information**  
Select language of the diagram

Diagram Language:

☐ SysML

☒ UML

☐ Profile

Figure 6.4: Selecting the diagram type

## 6. Evaluation

### Initialization information

Select name and kind of the diagram

The dialog box is titled 'Initialization information' and contains the following elements:

- Diagram Name:** A text input field containing 'Refactoring example'.
- Select a Diagram Kind:** A list of UML diagram types with checkboxes:
  - ☐ UML StateMachine Diagram
  - ☒ UML Activity Diagram (highlighted in orange)
  - ☐ UML Communication Diagram
  - ☐ UML UseCase Diagram
  - ☐ UML Composite Structure Diagram
  - ☐ UML Sequence Diagram
  - ☐ UML Package Diagram
  - ☐ UML Class Diagram
- You can load a template:** A section with a checkbox:
  - ☐ A UML model with basic primitive types (ModelWithBasicTypes.uml)
- ☐ Remember current selection
- Navigation buttons:** A question mark icon, '< Back', 'Next >', 'Cancel', and 'Finish'.

Figure 6.5: Adding an activity diagram to the project.

## 6.1. The example activity diagram

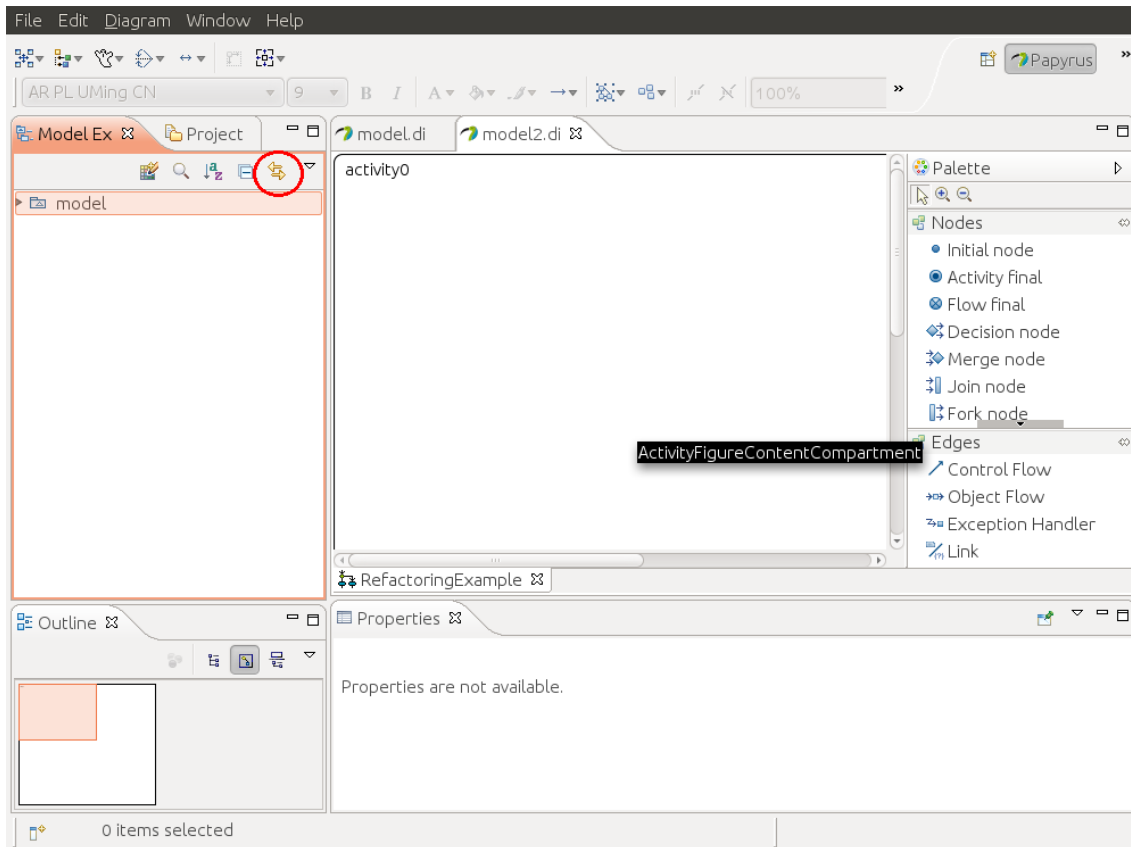


Figure 6.6: The Papyrus workbench

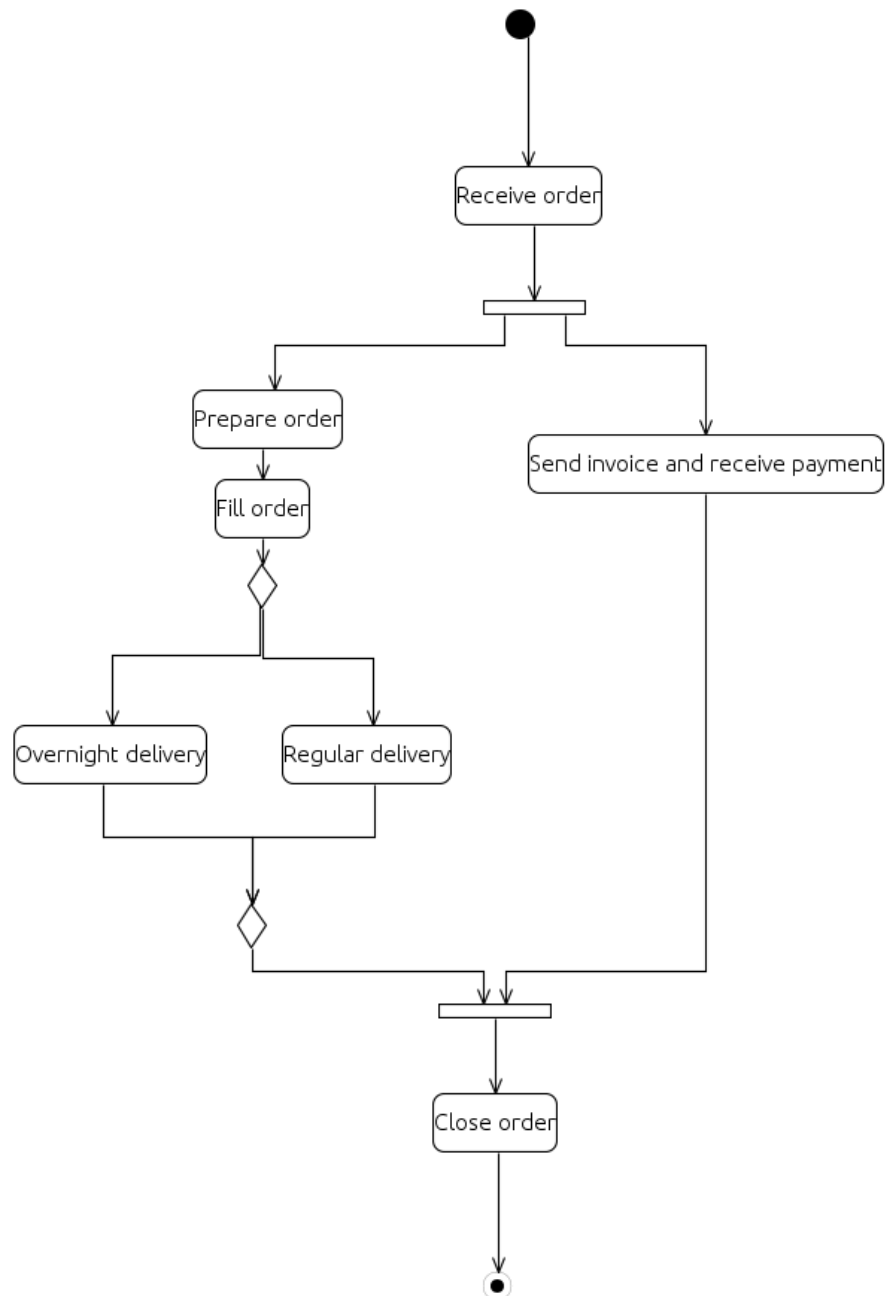


Figure 6.7: The example activity diagram

## 6.2. The Merge actions refactoring

In this section the *Merge actions* refactoring implementation will be executed on the example diagram created in section 6.1 and the results analyzed. When the diagram



in Figure 6.7 is examined, it can be seen that two actions seem to be doing more or less the same thing; the *Prepare order* action and the *Fill order* action, so they should be merged into one action.

The two actions are selected and the context menu is opened as shown in Figure 6.8. Only the Merge actions refactoring will be available because two objects have been selected, as described in section 5.7.2.

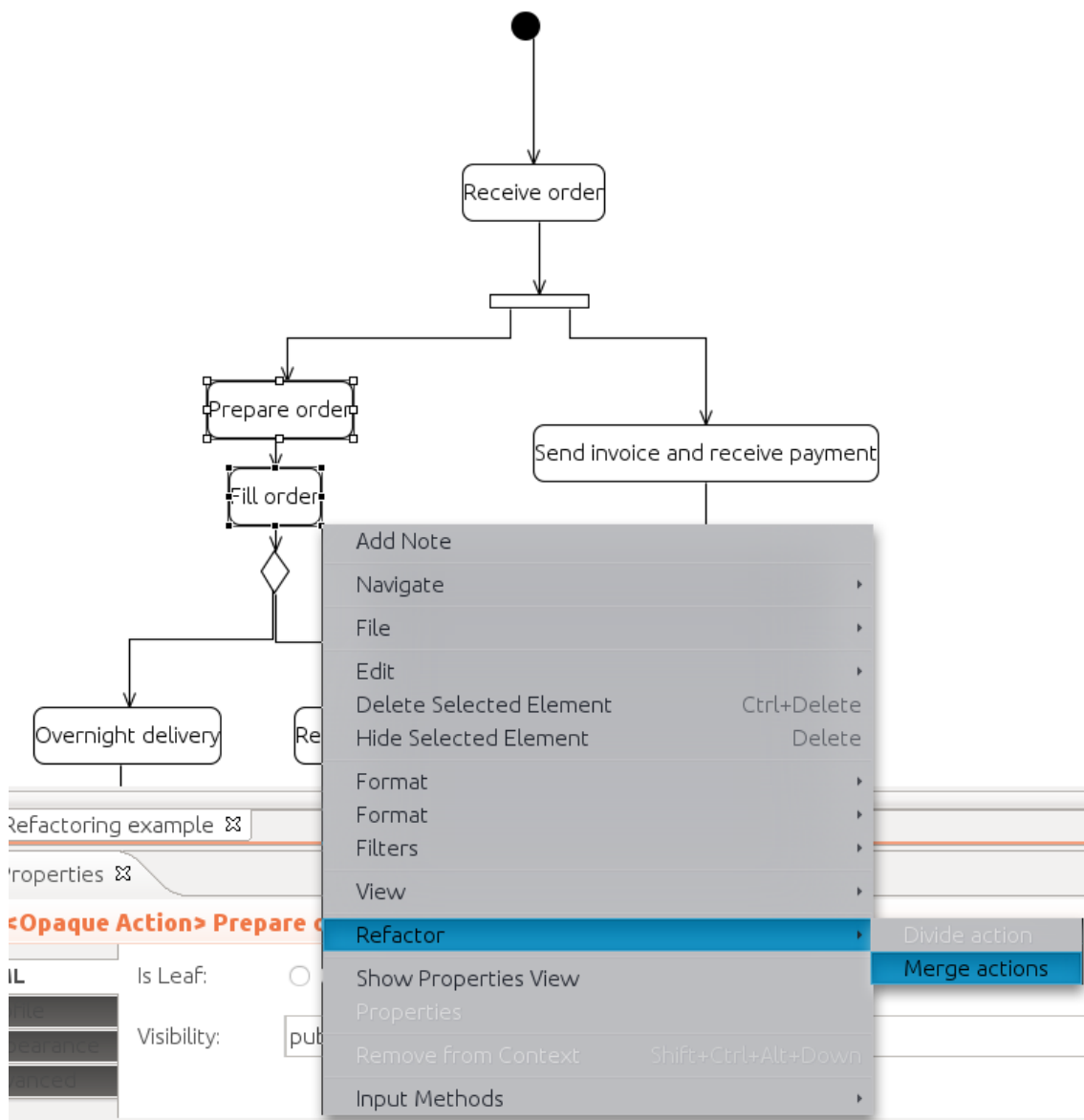


Figure 6.8: The Merge actions refactoring selected

When the refactoring has been selected and executed a dialog pops up telling the user that the underlying model has been changed due to the applied transformations

## 6. Evaluation

as shown in Figure 6.9, and asks the user if the diagram should be updated.

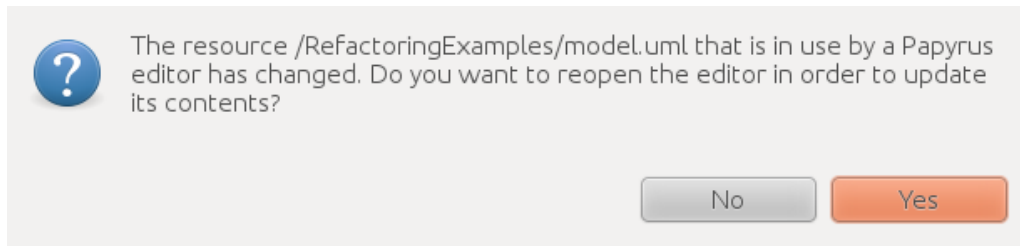


Figure 6.9: A dialog telling the user that the model resource has changed.

When the diagram has been updated based on the refactored models, the actions have been merged as can be seen in Figure 6.10. The *the Merge actions* transformation has merged the two selected actions into one action. The merged actions holds the name of the selected actions, *Prepare order - Fill order* and has the same coordinates as the *Fill order* action had originally. It can also be seen that the control flow edge between the original actions has been removed and the incoming control flow to the *Prepare order* action now points to the merged action.

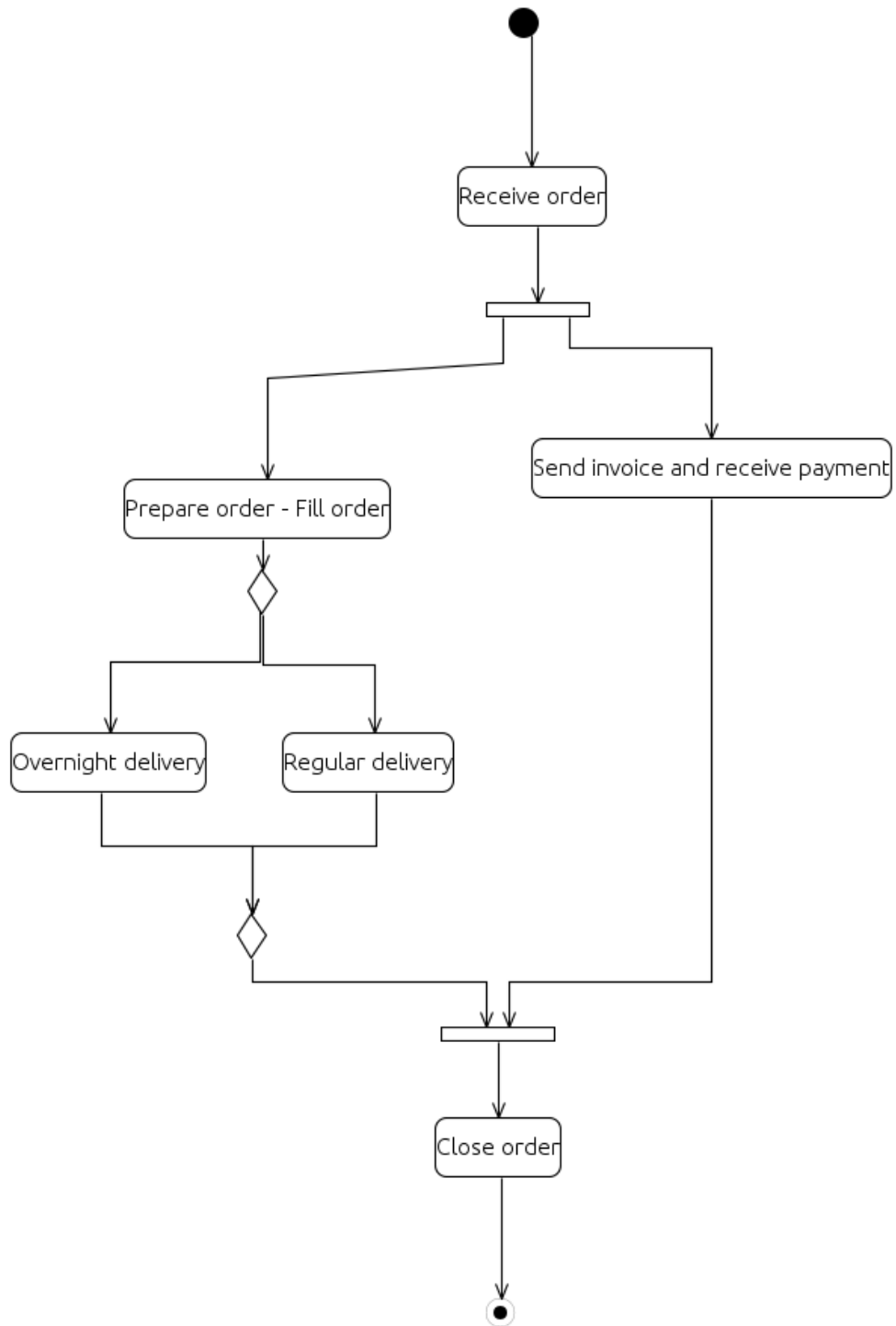


Figure 6.10: The diagram after the merge actions refactoring.

### 6.2.1. The UML model comparison

The underlying models should also be analyzed. The resulting UML model compared to the original in section 6.2.1 and the Notation model is compared to the original in section 6.2.2. As the models are quite large and contain many lines that do not change after the refactoring, only the relevant parts of each model will be shown in the comparison, with changes highlighted.

As seen when the models are compared, the changes are highlighted. Lines 16 and 30 in the original model have been deleted in the resulting model and lines 9 and 20 have changed according to the changes in the diagram. The node in line 9 has a new name, *Prepare order - Fill order* and a different incoming edge. The edge in line 20 in the original model has a new target in the resulting model, which means that the outgoing edge from the removed action is now connected to the merged action.

```

1 <?xml version="1.0" encoding="UTF-8"?>

9   <node xmi:type="uml:OpaqueAction" xmi:id="__kM_wHPiEeCn-pOPqs82NA"
   name="Fill order" outgoing="__mdeWcHPmEeCn-pOPqs82NA"
   incoming="__dzDnMHPqEeCn-pOPqs82NA"/>

15  <node xmi:type="uml:OpaqueAction" xmi:id="_R5yP8HPmEeCn-pOPqs82NA"
   name="Close order" outgoing="__ytcTQHPmEeCn-pOPqs82NA" incoming=
   "__yTxPAHPmEeCn-pOPqs82NA"/>
16  <node xmi:type="uml:OpaqueAction" xmi:id="_RnfCMHPqEeCn-pOPqs82NA"
   name="Prepare order" outgoing="__dzDnMHPqEeCn-pOPqs82NA"
   incoming="__iNERMHPmEeCn-pOPqs82NA"/>
17  <edge xmi:type="uml:ControlFlow" xmi:id="_giGkMHPmEeCn-pOPqs82NA"
   name="ControlFlow1" source="__0MLXkHPiEeCn-pOPqs82NA" target="
   "__6ryfUHPiEeCn-pOPqs82NA"/>
18  <edge xmi:type="uml:ControlFlow" xmi:id="_hDR-oHPmEeCn-pOPqs82NA"
   name="ControlFlow2" source="__6ryfUHPiEeCn-pOPqs82NA" target="
   "__8tPZsHPiEeCn-pOPqs82NA"/>
19  <edge xmi:type="uml:ControlFlow" xmi:id="_hpY8YHPmEeCn-pOPqs82NA"
   name="ControlFlow3" source="__8tPZsHPiEeCn-pOPqs82NA" target="
   "__7qgxYHPiEeCn-pOPqs82NA"/>
20  <edge xmi:type="uml:ControlFlow" xmi:id="_iNERMHPmEeCn-pOPqs82NA"
   name="ControlFlow4" source="__8tPZsHPiEeCn-pOPqs82NA"
   target="__RnfCMHPqEeCn-pOPqs82NA"/>

29  <edge xmi:type="uml:ControlFlow" xmi:id="_ytcTQHPmEeCn-pOPqs82NA"
   name="ControlFlow14" source="__R5yP8HPmEeCn-pOPqs82NA" target="
   "__2KlIAHPiEeCn-pOPqs82NA"/>
30  <edge xmi:type="uml:ControlFlow" xmi:id="_dzDnMHPqEeCn-pOPqs82NA"
   name="ControlFlow15" source="__RnfCMHPqEeCn-pOPqs82NA"
   target="__kM_wHPiEeCn-pOPqs82NA"/>
31 </packagedElement>
32 </uml:Model>

```

Listing 6.1: The UML model before refactoring

## 6. Evaluation

```
1 <?xml version="1.0" encoding="UTF-8"?>

9 <node xmi:type="uml:OpaqueAction" xmi:id="__kM_wHPmEeCn-pOPqs82NA"
  name="Prepare order - Fill order" outgoing="_mdeWcHPmEeCn-pOPqs82NA"
  incoming="_iNERMHPmEeCn-pOPqs82NA"/>

15 <node xmi:type="uml:OpaqueAction" xmi:id="_R5yP8HPmEeCn-pOPqs82NA"
  name="Close order" outgoing="_ytcTQHPmEeCn-pOPqs82NA" incoming=
  "_yTxPAHPmEeCn-pOPqs82NA"/>
16 <edge xmi:type="uml:ControlFlow" xmi:id="_giGkMHPmEeCn-pOPqs82NA"
  name="ControlFlow1" source="_0MLXkHPmEeCn-pOPqs82NA" target="
  _6ryfUHHPmEeCn-pOPqs82NA"/>
17 <edge xmi:type="uml:ControlFlow" xmi:id="_hDR-oHPmEeCn-pOPqs82NA"
  name="ControlFlow2" source="_6ryfUHHPmEeCn-pOPqs82NA" target="
  _8tPZsHPmEeCn-pOPqs82NA"/>
18 <edge xmi:type="uml:ControlFlow" xmi:id="_hpY8YHPmEeCn-pOPqs82NA"
  name="ControlFlow3" source="_8tPZsHPmEeCn-pOPqs82NA" target="
  _7qgxYHPmEeCn-pOPqs82NA"/>
19 <edge xmi:type="uml:ControlFlow" xmi:id="_iNERMHPmEeCn-pOPqs82NA"
  name="ControlFlow4" source="_8tPZsHPmEeCn-pOPqs82NA"
  target="__kM_wHPmEeCn-pOPqs82NA"/>

28 <edge xmi:type="uml:ControlFlow" xmi:id="_ytcTQHPmEeCn-pOPqs82NA"
  name="ControlFlow14" source="_R5yP8HPmEeCn-pOPqs82NA" target="
  _2KlIAHPmEeCn-pOPqs82NA"/>
29 </packageElement>
30 </uml:Model>
```

*Listing 6.2: The resulting UML model after the Merge actions refactoring*

### 6.2.2. The Notation model comparison

When the Notation models are compared, a few highlighted changes can be seen. The shape element in lines 223-237 has been removed in the resulting model along with the edge in lines 472-488. The edge in line 297 has changed so the target points the action which the removed shape was originally connected to.

```

1 <?xml version="1.0" encoding="UTF-8"?>

221     <layoutConstraint xmi:type="notation:Bounds" xmi:id="
222         _R5zeEnPmEeCn-pOPqs82NA" x="321" y="737"/>
223 </children>
224 <children xmi:type="notation:Shape" xmi:id="_RnhecHPqEeCn-pOPqs82NA"
225     type="3007" fontName="Ubuntu" fontHeight="11" lineColor="0">

235     <element xmi:type="uml:OpaqueAction"
236         href="model.uml#_RnfCMHPqEeCn-pOPqs82NA"/>
237     <layoutConstraint xmi:type="notation:Bounds"
238         xmi:id="_RniFgXPqEeCn-pOPqs82NA" x="160" y="257"/>
239 </children>
240 <layoutConstraint xmi:type="notation:Bounds" xmi:id="
241     _kN_cnXPIEeCn-pOPqs82NA" />
242 </children>

297 <edges xmi:type="notation:Connector" xmi:id="_iNGtcHPmEeCn-pOPqs82NA"
298     type="4004" source="_8tRO4HPiEeCn-pOPqs82NA"
299     target="_RnhecHPqEeCn-pOPqs82NA" lineColor="0">

469 <element xmi:type="uml:ControlFlow" href="model.uml#_ytcTQHPmEeCn-
470     pOPqs82NA"/>
471 <bendpoints xmi:type="notation:RelativeBendpoints" xmi:id="
472     _ytfWknPmEeCn-pOPqs82NA" points="[4, 20, -11, -48]$, [5, 68, -10,
473     0]" />
474 </edges>
475 <edges xmi:type="notation:Connector" xmi:id="_dzJt0HPqEeCn-pOPqs82NA"
476     type="4004" source="_RnhecHPqEeCn-pOPqs82NA"
477     target="__kON4HPiEeCn-pOPqs82NA" lineColor="0">

486 <element xmi:type="uml:ControlFlow"
487     href="model.uml#_dzDnMHPqEeCn-pOPqs82NA"/>
488 <bendpoints xmi:type="notation:RelativeBendpoints"
489     xmi:id="_dzKU4HPqEeCn-pOPqs82NA"
490     points="[22, 20, -54, -70]$, [108, 84, 32, -6]" />
491 </edges>
492 </notation:Diagram>

```

Listing 6.3: The Notation model before the refactoring

```

1 <?xml version="1.0" encoding="UTF-8"?>

221     <layoutConstraint xmi:type="notation:Bounds" xmi:id="
222         _R5zeEnPmEeCn-pOPqs82NA" x="321" y="737"/>
223     </children>
224     <layoutConstraint xmi:type="notation:Bounds" xmi:id="
225         _kN_cnXPIEeCn-pOPqs82NA"/>
226     </children>
227     <element xmi:type="uml:Activity" href="model.uml#_kHP7YHPmEeCn-
        pOPqs82NA"/>
228     <layoutConstraint xmi:type="notation:Bounds" xmi:id="_kN_cnnPIEeCn-
        pOPqs82NA" y="15"/>
229 </children>

282 <edges xmi:type="notation:Connector" xmi:id="_iNGtcHPmEeCn-pOPqs82NA"
    type="4004" source="_8tRO4HPmEeCn-pOPqs82NA"
    target="__kON4HPmEeCn-pOPqs82NA" lineColor="0">

454 <element xmi:type="uml:ControlFlow" href="model.uml#_ytTQHPmEeCn-
    pOPqs82NA"/>
455 <bendpoints xmi:type="notation:RelativeBendpoints" xmi:id="
    _ytfWknPmEeCn-pOPqs82NA" points="[4, 20, -11, -48]$, [5, 68, -10,
    0]"/>
456 </edges>
457 </notation:Diagram>

```

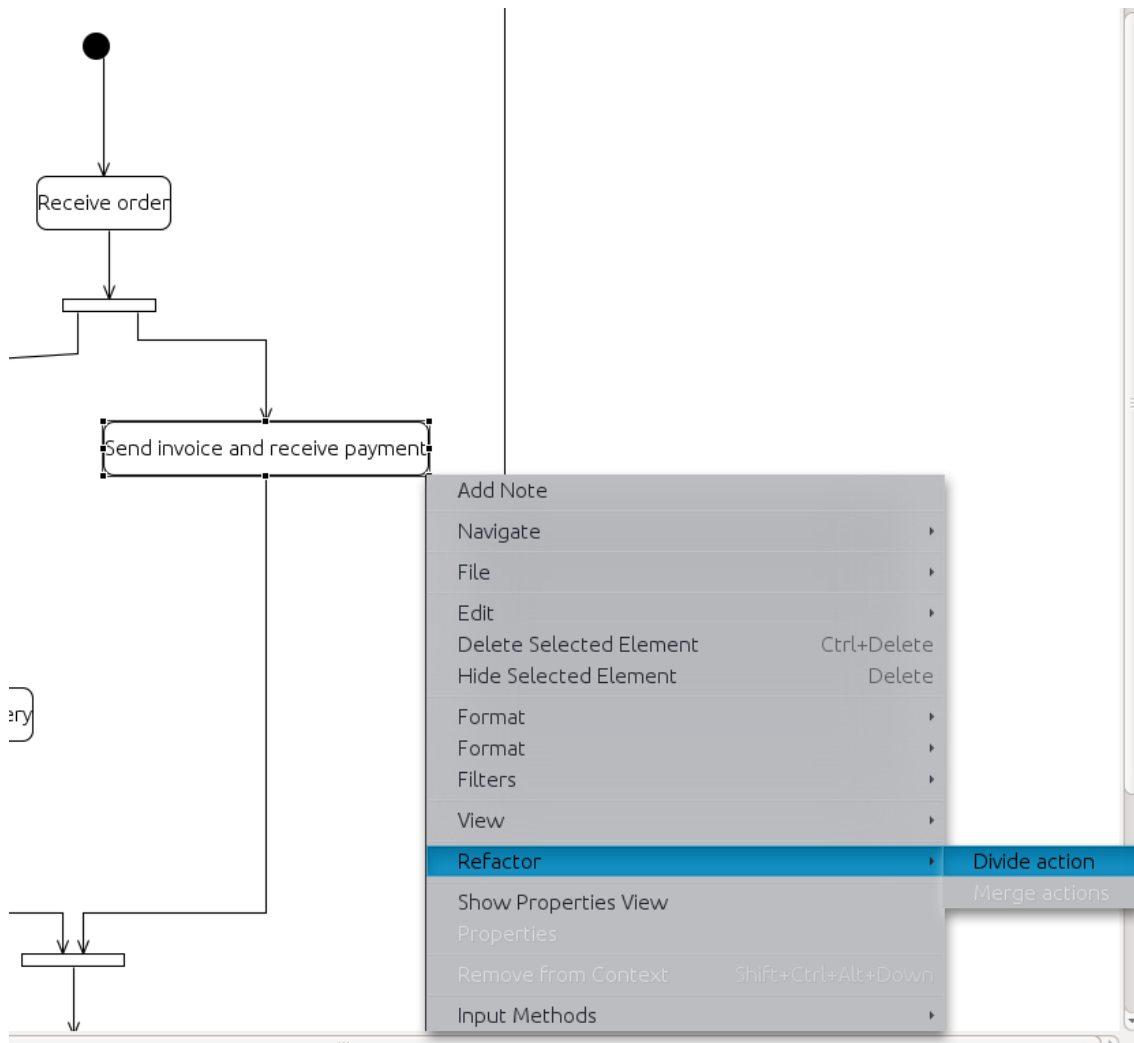
Listing 6.4: The resulting Notation model after the Merge actions refactoring

### 6.3. The Divide actions refactoring

In this section the *Divide actions* refactoring implementation will be executed on the example diagram which has already been refactored once in section 6.2 and the results analyzed. When the diagram in Figure 6.10 is examined, it can be seen that the *Send invoice and receive payment* action is doing too much to be just one action, so it should be refactored by dividing it into two actions.

The action to be divided is selected and the context menu is opened as shown in Figure 6.11. Only the *Divide action* refactoring will be available because one object has been selected, as described in section 5.7.2.





*Figure 6.11: The Divide actions refactoring selected*

When the refactoring has been selected and executed, the dialog telling the user that the underlying model has been changed, pops up and asks the user if the model should be updated, as in section 6.2.

When the diagram has been updated based on the refactored models, the action has been divided as can be seen in Figure 6.12.

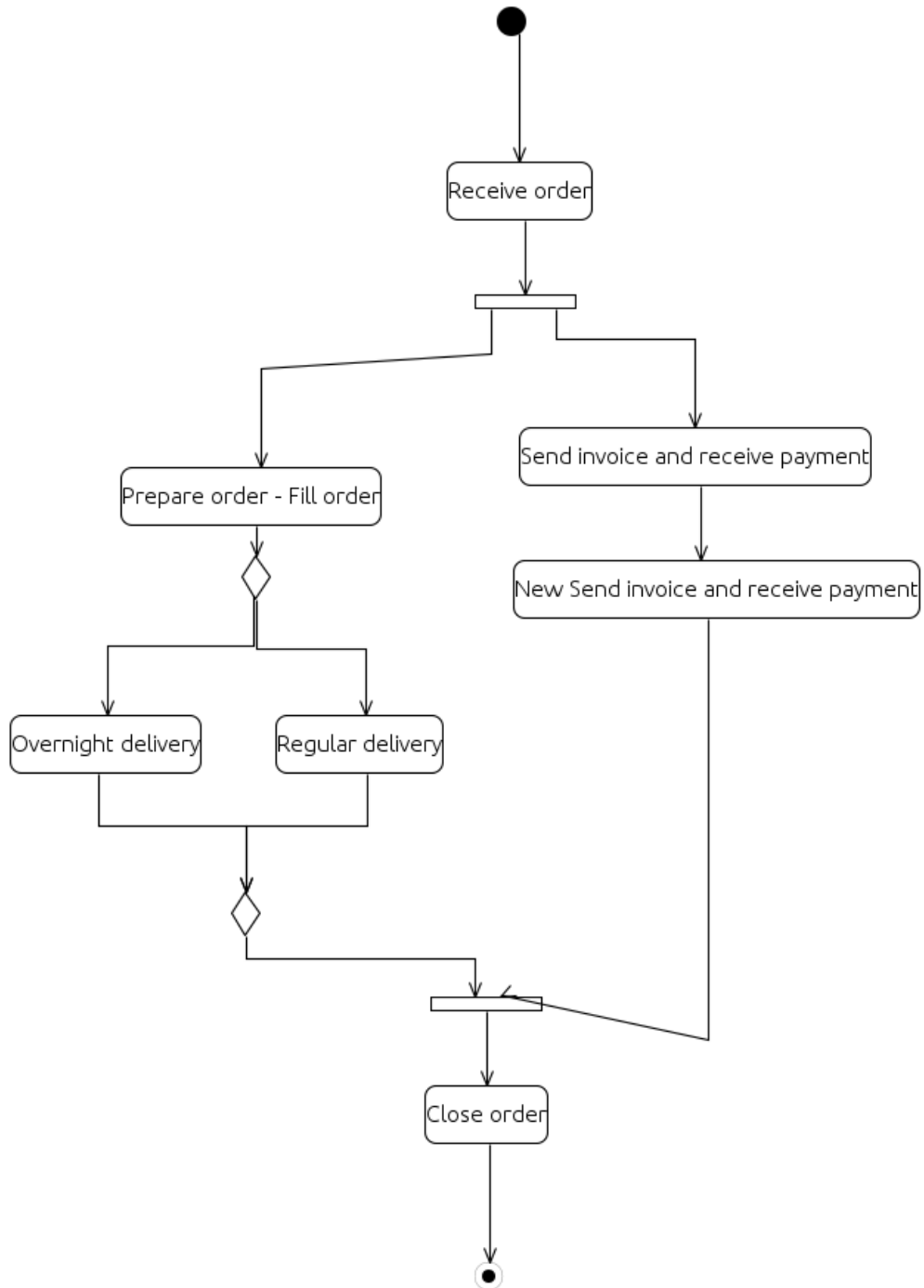


Figure 6.12: The diagram after the divide actions refactoring.

As Figure 6.12 demonstrates, the *Divide action* transformation has divided the selected action into two actions. The new action holds the name of the selected action with the "New" word prepended, *New Send invoice and receive payment*. It can also be seen that a control flow edge between the actions has been added and the

outgoing control flow from the *Send invoice and receive payment* action now starts at the new action. It should also be noted that the name of the new action is not very descriptive so the user might want to rename the action along with fixing the outgoing edge which crosses through the join node. This cleanup process will be described in section 6.4.

#### 6.3.1. The UML model comparison

The underlying models should also be analyzed. The resulting UML model compared to the original in section 6.3.1 and the Notation model is compared to the original in section 6.3.2. As the models are quite large and contain many lines that do not change after the refactoring, only the relevant parts of each model will be shown in the comparison, with changes highlighted.

As seen when the UML models are compared, there are a few highlighted changes. The node element in line 7 has another outgoing edge in the resulting model, the one which has been added in line 30 in the resulting model. A new action, *New Send invoice and receive payment*, has also been added in line 16 in the resulting model and the edge in line 26 in the original model has had its source changed to the new action.

## 6. Evaluation

```
1 <?xml version="1.0" encoding="UTF-8"?>

7   <node xmi:type="uml:OpaqueAction" xmi:id="_7qgxYHPmEeCn-pOPqs82NA"
   name="Send invoice and receive payment"
   outgoing="_x3kbMHPmEeCn-pOPqs82NA"
   incoming="_hpY8YHPmEeCn-pOPqs82NA"/>

15  <node xmi:type="uml:OpaqueAction" xmi:id="_R5yP8HPmEeCn-pOPqs82NA"
   name="Close order" outgoing="_ytcTQHPmEeCn-pOPqs82NA" incoming=
16  "_yTxPAHPmEeCn-pOPqs82NA"/>
   <edge xmi:type="uml:ControlFlow" xmi:id="_giGkMHPmEeCn-pOPqs82NA"
   name="ControlFlow1" source="_0MLXkHPmEeCn-pOPqs82NA" target="
   _6ryfUHPlEeCn-pOPqs82NA"/>

26  <edge xmi:type="uml:ControlFlow" xmi:id="_x3kbMHPmEeCn-pOPqs82NA"
   name="ControlFlow12" source="_7qgxYHPmEeCn-pOPqs82NA"
   target="_RSto4HPmEeCn-pOPqs82NA"/>

27  <edge xmi:type="uml:ControlFlow" xmi:id="_yTxPAHPmEeCn-pOPqs82NA"
   name="ControlFlow13" source="_RSto4HPmEeCn-pOPqs82NA" target="
   _R5yP8HPmEeCn-pOPqs82NA"/>
28  <edge xmi:type="uml:ControlFlow" xmi:id="_ytcTQHPmEeCn-pOPqs82NA"
   name="ControlFlow14" source="_R5yP8HPmEeCn-pOPqs82NA" target="
   _2KllAHPmEeCn-pOPqs82NA"/>
29 </packageElement>
30 </uml:Model>
```

Listing 6.5: The UML model before refactoring

```

1 <?xml version="1.0" encoding="UTF-8"?>

7   <node xmi:type="uml:OpaqueAction" xmi:id="_7qgxYHPIEeCn-pOPqs82NA"
   name="Send invoice and receive payment"
   outgoing="_BHCK0XPuEeCn-pOPqs82NA"
   incoming="_hpY8YHPmEeCn-pOPqs82NA"/>

15  <node xmi:type="uml:OpaqueAction" xmi:id="_R5yP8HPmEeCn-pOPqs82NA"
   name="Close order" outgoing="_ytcTQHPmEeCn-pOPqs82NA" incoming=
16  " _yTxPAHPmEeCn-pOPqs82NA" />
   <node xmi:type="uml:OpaqueAction" xmi:id="_BHCK0HPuEeCn-pOPqs82NA"
   name="New Send invoice and receive payment"
   outgoing="_x3kbMHPmEeCn-pOPqs82NA"
   incoming="_BHCK0XPuEeCn-pOPqs82NA"/>
17  <edge xmi:type="uml:ControlFlow" xmi:id="_giGkMHPmEeCn-pOPqs82NA"
   name="ControlFlow1" source="_0MLXkHPIEeCn-pOPqs82NA" target=
   "_6ryfUHPIEeCn-pOPqs82NA" />

27  <edge xmi:type="uml:ControlFlow" xmi:id="_x3kbMHPmEeCn-pOPqs82NA"
   name="ControlFlow12" source="_BHCK0HPuEeCn-pOPqs82NA"
   target="_RSto4HPmEeCn-pOPqs82NA"/>
28  <edge xmi:type="uml:ControlFlow" xmi:id="_yTxPAHPmEeCn-pOPqs82NA"
   name="ControlFlow13" source="_RSto4HPmEeCn-pOPqs82NA" target=
   "_R5yP8HPmEeCn-pOPqs82NA" />
29  <edge xmi:type="uml:ControlFlow" xmi:id="_ytcTQHPmEeCn-pOPqs82NA"
   name="ControlFlow14" source="_R5yP8HPmEeCn-pOPqs82NA" target=
   "_2KllAHPIEeCn-pOPqs82NA" />
30  <edge xmi:type="uml:ControlFlow" xmi:id="_BHCK0XPuEeCn-pOPqs82NA"
   name="New Edge" source="_7qgxYHPIEeCn-pOPqs82NA"
   target="_BHCK0HPuEeCn-pOPqs82NA"/>
31 </packagedElement>
32 </uml:Model>

```

Listing 6.6: The resulting UML model from the Divide actions refactoring

### 6.3.2. The Notation model comparison

As seen when the Notation models are compared, there are a few highlighted changes. The edge in line 406 in the original model has a new source which is the new shape element added in lines 223-237 in the resulting model. An edge has also been added in lines 472-489 in the resulting model which goes from the original action and to the newly added action.

```

1 <?xml version="1.0" encoding="UTF-8"?>

220     <element xmi:type="uml:OpaqueAction" href="model.uml#
221         _R5yP8HPmEeCn-pOPqs82NA"/>
221     <layoutConstraint xmi:type="notation:Bounds" xmi:id="
222         _R5zeEnPmEeCn-pOPqs82NA" x="321" y="737"/>
222 </children>
223     <layoutConstraint xmi:type="notation:Bounds" xmi:id="
224         _kN_cnXPIEeCn-pOPqs82NA"/>
224 </children>

406 <edges xmi:type="notation:Connector" xmi:id="_x3mQYHPmEeCn-pOPqs82NA"
    type="4004" source="_7qh_gHPmEeCn-pOPqs82NA"
    target="_RSzIcHPmEeCn-pOPqs82NA" lineColor="0">

454     <element xmi:type="uml:ControlFlow" href="model.uml#_ytcTQHPmEeCn-
455         pOPqs82NA"/>
455     <bendpoints xmi:type="notation:RelativeBendpoints" xmi:id="
456         _ytfWknPmEeCn-pOPqs82NA" points="[4, 20, -11, -48]$, [5, 68, -10,
457         0]"/>
456 </edges>
457 </notation:Diagram>

```

*Listing 6.7: The Notation model before refactoring*

```

1 <?xml version="1.0" encoding="UTF-8"?>

220 <element xmi:type="uml:OpaqueAction" href="model.uml#
    _R5yP8HPmEeCn-pOPqs82NA"/>
221 <layoutConstraint xmi:type="notation:Bounds" xmi:id="
    _R5zeEnPmEeCn-pOPqs82NA" x="321" y="737"/>
222 </children>
223 <children xmi:type="notation:Shape" xmi:id="_SpltElmLki9rsGaovxt803"
    type="3007" fontName="Ubuntu" fontHeight="11" lineColor="0">

235 <element xmi:type="uml:OpaqueAction"
    href="model.uml#_BHCK0HPuEeCn-pOPqs82NA"/>
236 <layoutConstraint xmi:type="notation:Bounds"
    xmi:id="_4JlcyHPvEeCn-pOPqs82NA" x="381" y="377"/>
237 </children>
238 <layoutConstraint xmi:type="notation:Bounds" xmi:id="
    _kN_cnXPIEeCn-pOPqs82NA"/>
239 </children>

421 <edges xmi:type="notation:Connector" xmi:id="_x3mQYHPmEeCn-pOPqs82NA"
    type="4004" source="_SpltElmLki9rsGaovxt803"
    target="_RSzIcHPmEeCn-pOPqs82NA" lineColor="0">

469 <element xmi:type="uml:ControlFlow" href="model.uml#_ytcTQHPmEeCn-
    pOPqs82NA"/>
470 <bendpoints xmi:type="notation:RelativeBendpoints" xmi:id="
    _ytfWknPmEeCn-pOPqs82NA" points="[4, 20, -11, -48]$, [5, 68, -10,
    0]"/>
471 </edges>
472 <edges xmi:type="notation:Connector" xmi:id="_4JpHIHPvEeCn-pOPqs82NA"
    type="4004" source="_7qh_gHPiEeCn-pOPqs82NA"
    target="_SpltElmLki9rsGaovxt803" lineColor="0">

486 <element xmi:type="uml:ControlFlow"
    href="model.uml#_BHCK0XPuEeCn-pOPqs82NA"/>
487 <bendpoints xmi:type="notation:RelativeBendpoints"
    xmi:id="_4JpuNXpVeeCn-pOPqs82NA"
    points="[3, 20, -12, -70]$, [11, 70, -4, -20]"/>
488 <targetAnchor xmi:type="notation:IdentityAnchor"
    xmi:id="_-xI_kHPvEeCn-pOPqs82NA" id="(0.4601449275362319,0.025)"/>
489 </edges>
490 </notation:Diagram>

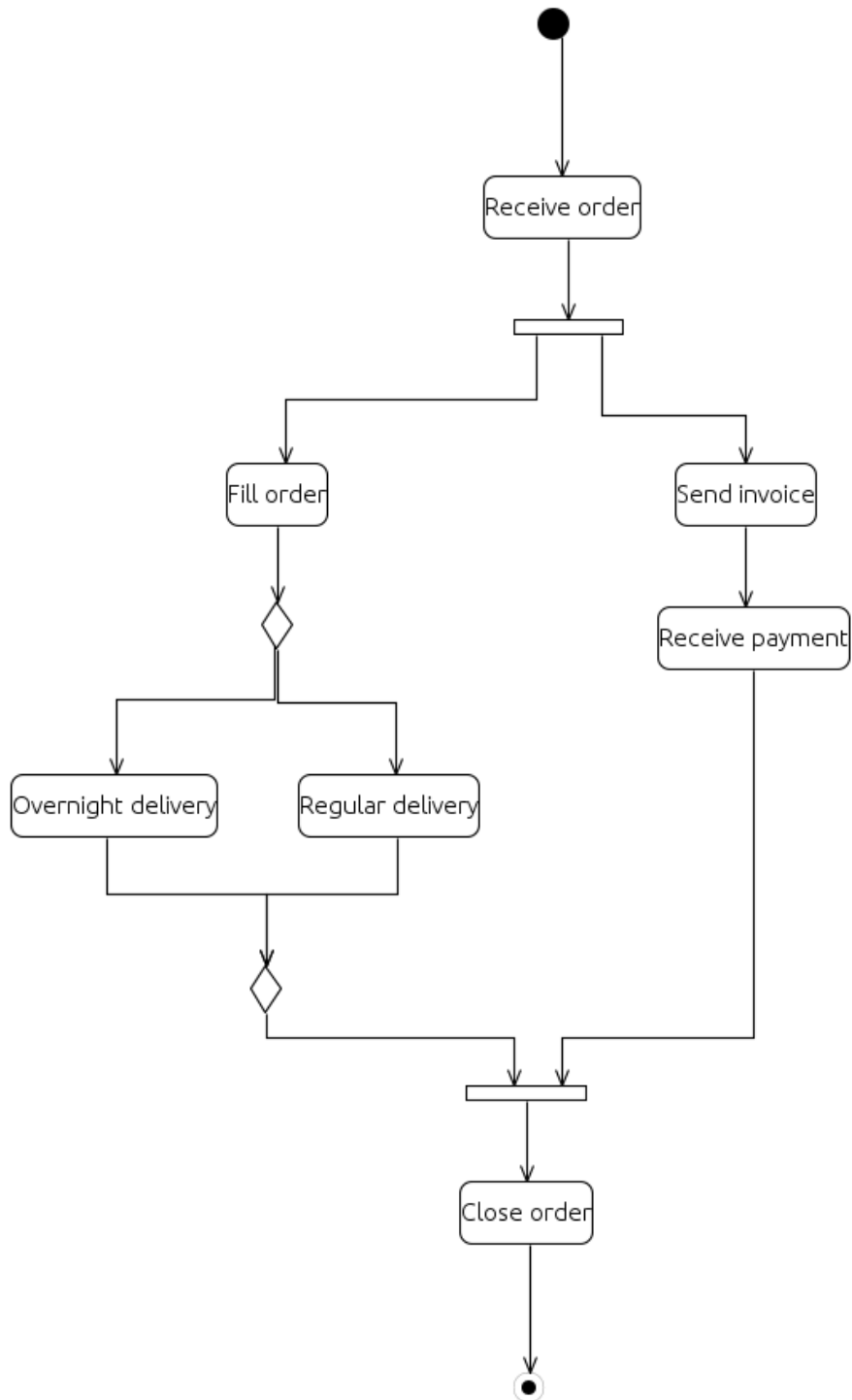
```

Listing 6.8: The resulting Notation model from the Divide action refactoring

## 6.4. The finishing touches

After the two refactorings have been executed on the model the user might want to change a few things. The name of the element which was merged in section 6.2 is a bit too descriptive so it should be renamed from *Prepare order - fill order* to *Fill order*. However, the actions divided in section 6.3 do not have very descriptive names. The *Send invoice and receive payment* action should be renamed to *Send invoice* and the *New Send invoice and receive payment* action should be renamed to *Receive payment*. For aesthetic purposes, the control flow edges should also be moved so they don't cross over any nodes and made straight. The resulting diagram is shown in Figure 6.13.



*Figure 6.13: The final model*

## 6.5. Discussion

As this chapter demonstrates, a UML model and its diagrammatic representation can be automatically refactored with high-level model-to-model transformations. In section 5.1 the difference between refactoring with model-to-model transformations and using a low-level solution is discussed.

It can be argued that using model-to-model transformations is a good choice because the transformations shown in this thesis are only approximately 100 lines of quite simple code, but would probably be quite more complex in a regular programming language where the XMI files or a similar internal data structure would be directly manipulated. The QVTO transformations are also not tool specific as opposed to a low-level solution which would need more adjustments when the metamodel changes. As a result, the QVTO transformation can be reused between different tools, excluding the blackboxing library which is specific to Eclipse. This also means that a QVTO transformation could be more easily adapted to another metamodel than a lower level solution. The diagrammatic representation is however proprietary to Papyrus, so the diagram transformations are not currently reusable in other UML editors without adjustment. If more UML editors would adopt the UMLDI as a standard format for storing UML models, refactorings would become more easily reused between editors.

There are some limitations to the research in this thesis, e.g. using QVTO will probably be slower than a low-level solution, especially on large models. However, this was not researched in the thesis. Another limitation might be concerning the expressiveness of QVTO, but as has been shown, when such problems arise they can be solved by calling Java code from QVTO using a blackboxing library.

When compared to earlier work in the field of refactoring UML models, the approach outlined in this thesis brings some new developments, especially refactoring on both the model level and the diagram level at the same time. The solutions to the problems listed in section 5.3 also provide a basis on which more refactorings can be developed for activity diagrams and other diagrams in the UML.

# 7. Conclusion

This chapter concludes the thesis with a summary in section 7.1 and an outlook in section 7.2.

## 7.1. Summary

Software refactoring is a powerful tool in the software developer's toolbox when battling software aging. As a part of model-driven development, it can also be a precious tool when developing software models. However, refactoring models is still mostly an academic subject and more work needs to be done before it will be used in the real world. As detailed in this thesis some work has already been done on refactoring UML models, however without refactoring on the diagram as well. Refactoring the diagrammatic representation of a model creates new problems which do not exist for refactoring code, like the positioning of objects which does not affect the meaning of the diagram but will affect the readability.

In this thesis, two refactorings for activity diagrams, *Merge actions* and *Divide action* have been developed and implemented in QVTO for the Papyrus UML editor in the Eclipse IDE. The refactorings change the structure of both the underlying UML model and the diagram model, where elements are added to the diagram, removed from it or just changed in place. This thesis is the first known work where this is described.

## 7.2. Outlook

Even though this thesis adds to the understanding and implementation of refactoring models and diagrams, there are a few things that still need work. A very interesting subject is researching how to make the automatic placement of elements more precise. An algorithm might be developed to decide the placement of new objects in the diagram. Another important task is to create a catalog of the refactorings

## 7. Conclusion

developed in this thesis and in other related work in one place and implement them based the method proposed in this thesis. As the study of related work reveals, there exist no refactorings for UML sequence diagrams, even though it is quite well known and used by professionals.

Some support mechanisms could also be added to the implemented refactoring tool: the ability to undo and redo transformations, and giving the user the possibility of reviewing the refactoring results before executing it. For other refactorings than those proposed in this thesis, there is also the possibility that an object in one diagram might be referenced in another, so the refactoring might look for references in other diagrams and reflect the changes there.

Refactorings are often used to combat code smells, which are symptoms in a program which indicate problems. If smells are defined for UML models, an automatic UML refactoring tool might suggest refactorings based on a smell. Based on that, defining UML smells and developing refactorings to fix the underlying problem is an interesting research subject.

The UML Diagram Interchange (UMLDI) is a well defined standard for exchanging information about UML models and their diagrammatic representation but not very widely used in practice. Despite that, it might be an interesting project to develop UML refactorings for the UMLDI using model-to-model transformations, just like it has been done for the proprietary Papyrus diagram model. If more UML editors adopt UMLDI, this would enable exchanging the QVTO diagram transformations between editors.

One other interesting possibility is refactoring the metalanguages themselves like QVT. As QVT is a programming language with a syntax which resembles many popular languages, QVT transformations themselves could be refactored in the same way as these languages, but the corresponding refactorings and tool support is needed.

# Acronyms

<b>AGG</b>	Attributed Graph Grammar System
<b>API</b>	Application Programming Interface
<b>ATL</b>	ATL Transformation Language
<b>CMOF</b>	Complete Meta-Object Facility
<b>EMF</b>	Eclipse Model Framework
<b>EMOF</b>	Essential Meta-Object Facility
<b>GEF</b>	Eclipse Graphical Editing Framework
<b>GMP</b>	Eclipse Graphical Modeling Project
<b>IDE</b>	Integrated Development Environment
<b>MDT</b>	Eclipse Model Development Tools
<b>MOF</b>	Meta-Object Facility
<b>OCL</b>	Object Constraint Language
<b>OMG</b>	Object Management Group
<b>QVT</b>	Query/View/Transformation
<b>QVTO</b>	Query/View/Transformation Operational
<b>UML</b>	Unified Modeling Language
<b>UMLDI</b>	UML Diagram Interchange
<b>URI</b>	Uniform Resource Identifier

<b>URL</b>	Uniform Resource Locator
<b>XMI</b>	XML Metadata Interchange
<b>XML</b>	Extensible Markup Language

# Bibliography

- [1] Dave Astels. Refactoring with UML. In *3rd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2002)*, pages 67–70, 2002.
- [2] Thomas Baar and Slaviša Marković. A graphical approach to prove the semantic preservation of UML/OCL refactoring rules. In *Proceedings of the 6th international Andrei Ershov memorial conference on Perspectives of systems informatics*, PSI'06, pages 70–83, Berlin, Heidelberg, 2007. Springer-Verlag.
- [3] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Precise semantics of EMF model transformations by graph transformation. In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, MoDELS '08, pages 53–67, Berlin, Heidelberg, 2008. Springer-Verlag.
- [4] Marko Boger, Thorsten Sturm, and Per Fragemann. Refactoring browser for UML. In *Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, NODE '02, pages 366–377, London, UK, UK, 2003. Springer-Verlag.
- [5] Dobrzański and Ludwik Kuźniarz. Practical refactoring of executable UML models. *Nordic J. of Computing*, 12:343–360, August 2005.
- [6] Alessandro Folli and Tom Mens. Refactoring of UML models using AGG. *ECEASST*, pages –1–1, 2007.
- [7] The Eclipse foundation. The Eclipse EMF website. <http://www.eclipse.org/modeling/emf/>, May 2011.
- [8] The Eclipse foundation. The Eclipse GEF website. <http://www.eclipse.org/gef/>, May 2011.
- [9] The Eclipse foundation. The Eclipse GMF website. <http://www.eclipse.org/modeling/gmf/>, May 2011.
- [10] The Eclipse foundation. The Eclipse help website. <http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.gmf.doc/prog-guide/runtime/Developer%20Guide%20to%20Diagram%20Runtime.html>, May 2011.

## BIBLIOGRAPHY

- [11] The Eclipse foundation. The Eclipse M2M website. <http://www.eclipse.org/m2m/>, May 2011.
- [12] The Eclipse foundation. The Eclipse model development tools website. <http://www.eclipse.org/modeling/mdt/?project=uml2>, May 2011.
- [13] The Eclipse foundation. The Eclipse model development tools website. <http://www.eclipse.org/modeling/mdt/?project=uml2tools>, May 2011.
- [14] The Eclipse foundation. The Eclipse Papyrus website. <http://www.eclipse.org/modeling/mdt/papyrus/>, May 2011.
- [15] The Eclipse foundation. The Eclipse QVT wiki. <http://wiki.eclipse.org/QVTOML/Examples/InvokeInJava>, May 2011.
- [16] The Eclipse foundation. The Eclipse Subversive website. <http://www.eclipse.org/subversive/>, May 2011.
- [17] The Eclipse foundation. The Eclipse website. <http://www.eclipse.org/org/>, May 2011.
- [18] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [19] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Edition)*. Addison-Wesley Professional, 3 edition, September 2003.
- [20] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: abstraction and reuse of object-oriented design*, pages 701–717. Springer-Verlag New York, Inc., New York, NY, USA, 2002.
- [21] Gentleware. The Gentleware website. <http://www.gentleware.com/new-poseidon-for-uml-8-0.html>, May 2011.
- [22] Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards automating source-consistent uml refactorings. In *In Proceedings of the 6th International Conference on UML - The Unified Modeling Language*, pages 144–158. Springer, 2003.
- [23] Reiko Heckel. Graph transformation in a nutshell. In *Electr. Notes Theor. Comput. Sci*, pages 187–198. Elsevier, 2006.
- [24] IBM. The IBM Rational Rhapsody website. [www.ibm.com/software/awdtools/rhapsody/](http://www.ibm.com/software/awdtools/rhapsody/), May 2011.
- [25] MagicDraw. The MagicDraw website. <http://www.magicdraw.com/>, May 2011.



- [26] Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.0. <http://www.omg.org/spec/QVT/1.0/PDF/>, April 2008.
- [27] Object Management Group (OMG). Object Constraint Language (OCL). Technical report, Object Management Group (OMG), 2010.
- [28] Object Management Group (OMG). XML Metadata Interchange (XMI) Specification. Technical report, Object Management Group (OMG), 2002.
- [29] Object Management Group (OMG). UML 2.0 Superstructure Specification. Technical report, Object Management Group (OMG), August 2005.
- [30] Object Management Group (OMG). *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006.
- [31] Object Management Group (OMG). UML 2.0 Diagram Interchange Specification. Technical Report pct/03-09-01, Object Management Group (OMG), 2006.
- [32] Object Management Group (OMG). OMG Unified Modeling Language (OMG UML), Infrastructure, V2.4. Technical report, Object Management Group (OMG), 2010.
- [33] Object Management Group (OMG). The OMG website. <http://www.omg.org/marketing/about-omg.htm>, May 2011.
- [34] William F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois, Champaign, IL, USA, 1992. UMI Order No. GAX93-05645.
- [35] David Lorge Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.
- [36] Ivan Porres, Turku Centre, and Computer Science. Model refactorings as rule-based update transformations. In *In Proceedings of UML 2003 Conference, Springer-Verlag LNCS 2863*, pages 159–174. Springer, 2003.
- [37] Björn Regnell. *Requirements Engineering with Use Cases - a Basis for Software Development*. PhD thesis, Lund University, 1999.
- [38] James M. Bieman Robert B. France. Multi-view software evolution: A UML-based framework for evolving object-oriented software. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ICSM '01, pages 386–, Washington, DC, USA, 2001. IEEE Computer Society.

## BIBLIOGRAPHY

- [39] Kexing Rui and Greg Butler. Refactoring use case models: the metamodel. In *Proceedings of the 26th Australasian computer science conference - Volume 16*, ACSC '03, pages 301–308, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.
- [40] Philipp Seuring. Design and implementation of a UML model refactoring tool. *Master's thesis, Hasso-Plattner-Institute for Software Systems Engineering at the University of Potsdam*, 2005.
- [41] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley Professional, 2 edition, January 2008.
- [42] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring uml models. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 134–148, London, UK, UK, 2001. Springer-Verlag.
- [43] Tigris. The Subclipse website. <http://subclipse.tigris.org/>, May 2011.
- [44] TOPCASED. The TOPCASED website. <http://topcased.org/>, May 2011.

# A. The UML refactoring plugin

In this appendix the implemented UML refactoring plugin for Papyrus is outlined in terms of its dependencies, where the code is obtainable and how it is installed. Section A.1 outlines the dependencies of the plugin and how they can be installed and section A.2 demonstrates how the code can be obtained through a version control repository. In section A.3 the installation of the plugin is demonstrated and section A.4 then concludes the chapter with a summary of how the code can be examined at runtime with a debugger.

## A.1. Dependencies

The UML refactoring plugin depends on version 3.6 of the Eclipse IDE along with a few extra plugins. The whole ecosystem needed is comprised of these parts:

- Eclipse IDE 3.6 (Helios)
- Java 1.6
- Eclipse Modeling Framework 2.6.1
- UML2 3.1.2
- Operational QVT for Eclipse 3.0.1
- Papyrus UML 0.7.0

When the Eclipse IDE 3.6 has been installed, the simplest way of installing the rest of the needed plugins is by installing the Eclipse Modeling Discovery UI through the *Install new software* option in Eclipse as shown in Figure A.1, along with the UML2 extender SDK.

## A. The UML refactoring plugin

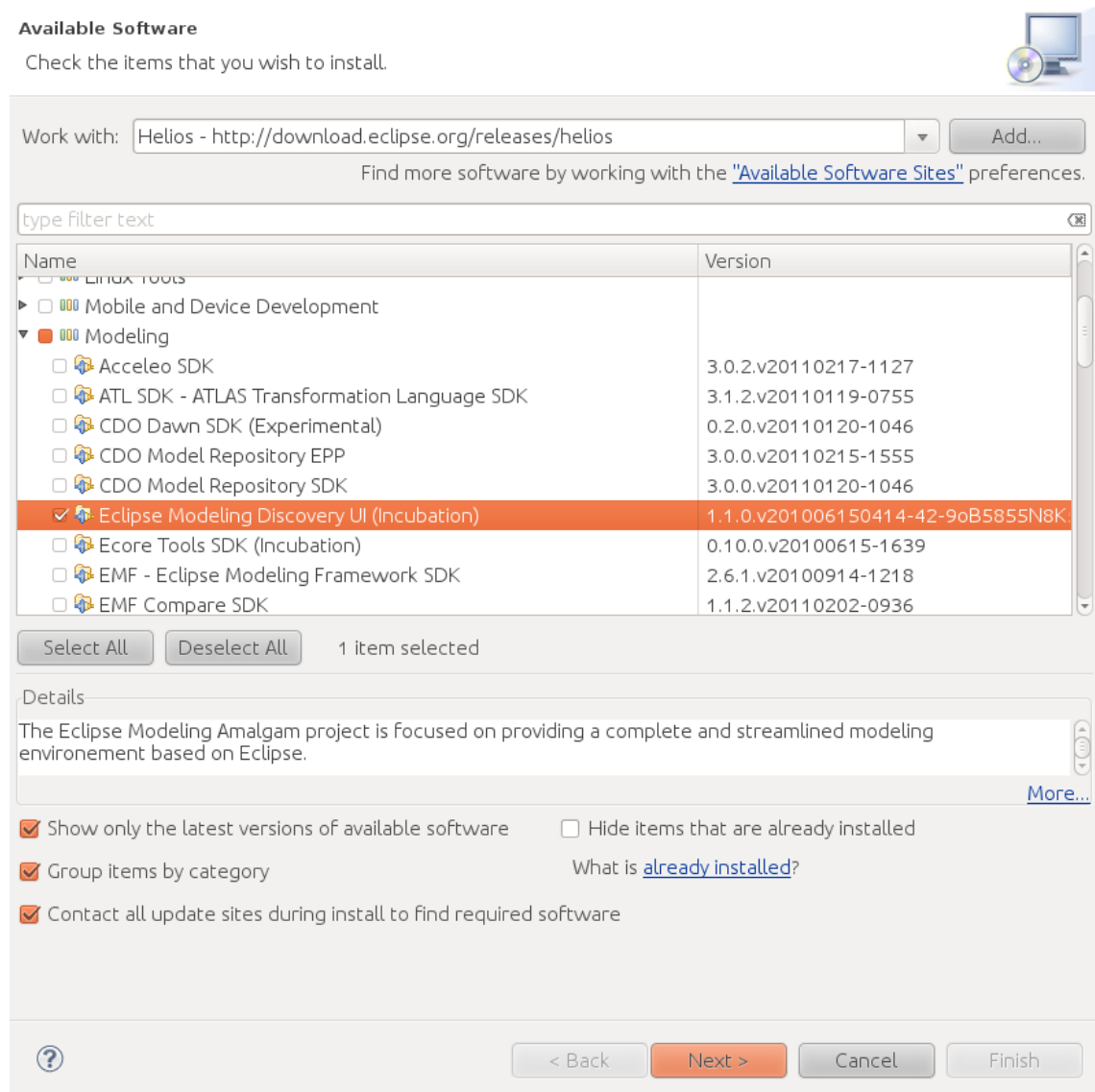
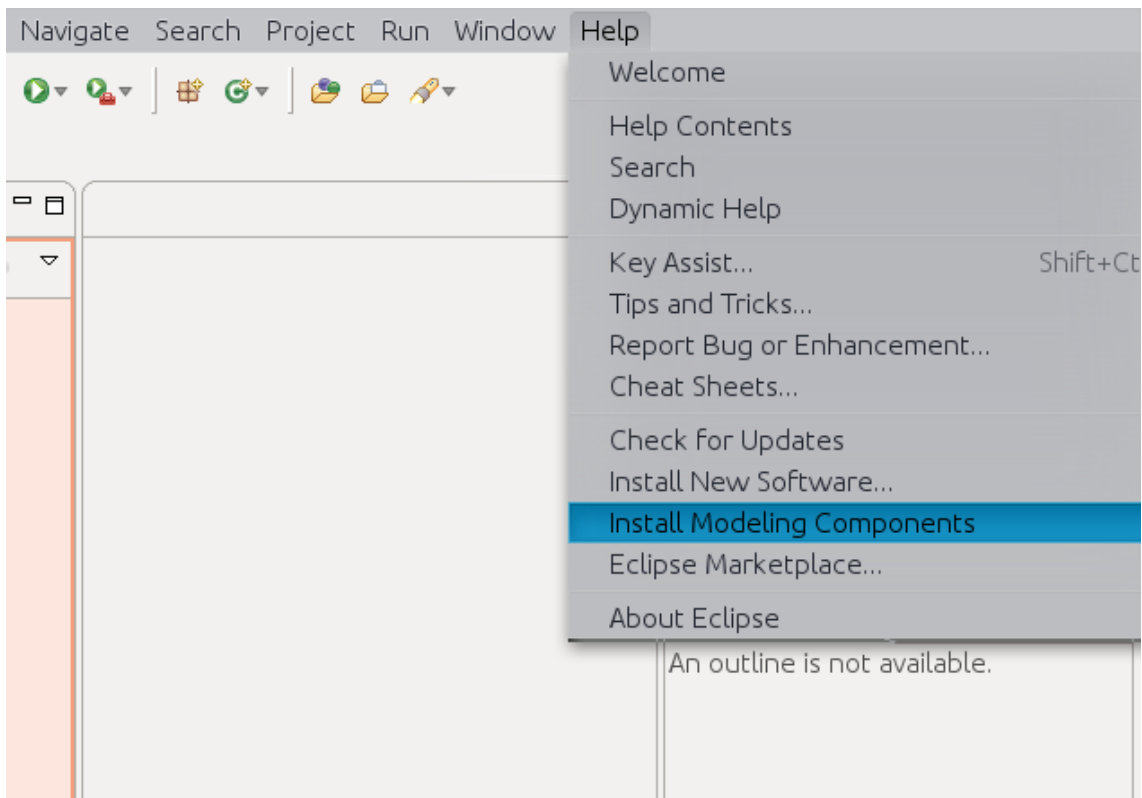


Figure A.1: Installing the Eclipse Modeling Discovery UI

When the Eclipse Modeling Discovery UI and the UML2 extender SDK have been installed, the rest of the modeling software can be installed with the *Install Modeling Components* option shown in Figure A.2.



*Figure A.2: Selecting the model installation UI*

In the Eclipse Modeling Discovery UI both the QVTO and Papyrus UML plugins can be selected and installed in one go, as shown in Figure A.3.

## A. The UML refactoring plugin

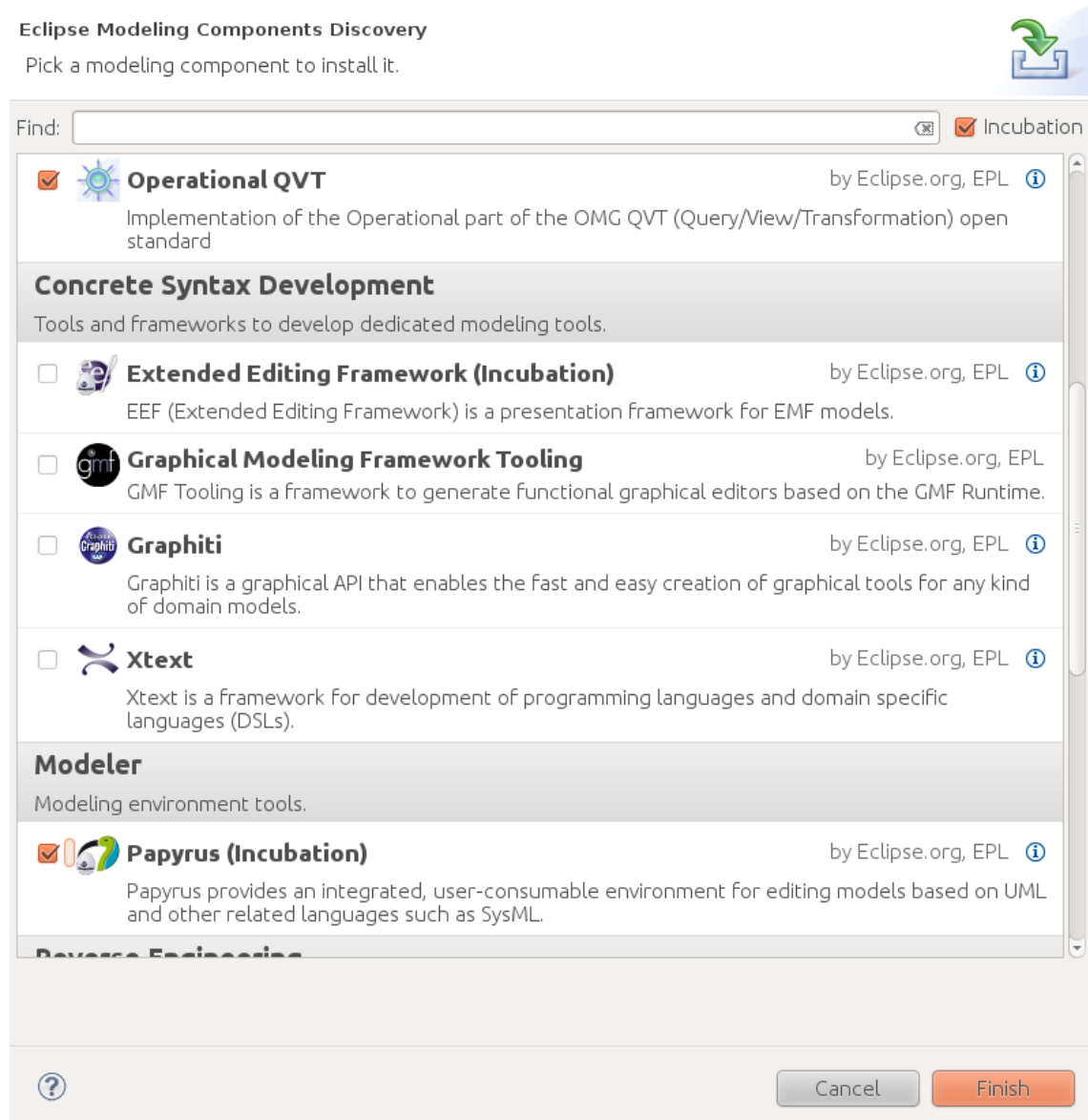


Figure A.3: Installing QVTO and Papyrus UML

## A.2. Obtaining the code from an SVN repository

The code behind the plugin is kept in a public SVN repository at the URL <http://svn2.xp-dev.com/svn/UMLrfp/>, so anyone can obtain it without logging in. A few SVN plugins exist for Eclipse, e.g. Subversive [16] and Subclipse [43], which can be used to import the code into Eclipse. In this demonstration the Subversive SVN plugin is used.

## A.2. Obtaining the code from an SVN repository

To obtain the code, a wizard for creating a new SVN project is started in Eclipse. The SVN repository URL is input in the URL field as shown in Figure A.4, and proceeded to the next step.

**Enter Repository Location Information**

Define the SVN repository location information. You can specify additional settings for proxy and svn+ssh, https connections.

General Advanced SSH Settings SSL Settings

URL:  Browse...

Label

☒ Use the repository URL as the label

☐ Use a custom label:

Authentication

User:

Password:

☐ Save authentication (could trigger secure storage login)

To manage your security data, please see ["Secure Storage"](#)

Show Credentials For:

☒ Validate Repository Location on finish

Reset Changes

? < Back Next > Cancel Finish

Figure A.4: Creating a new SVN project

In the next step the head revision option is selected as shown in Figure A.5.

## A. The UML refactoring plugin

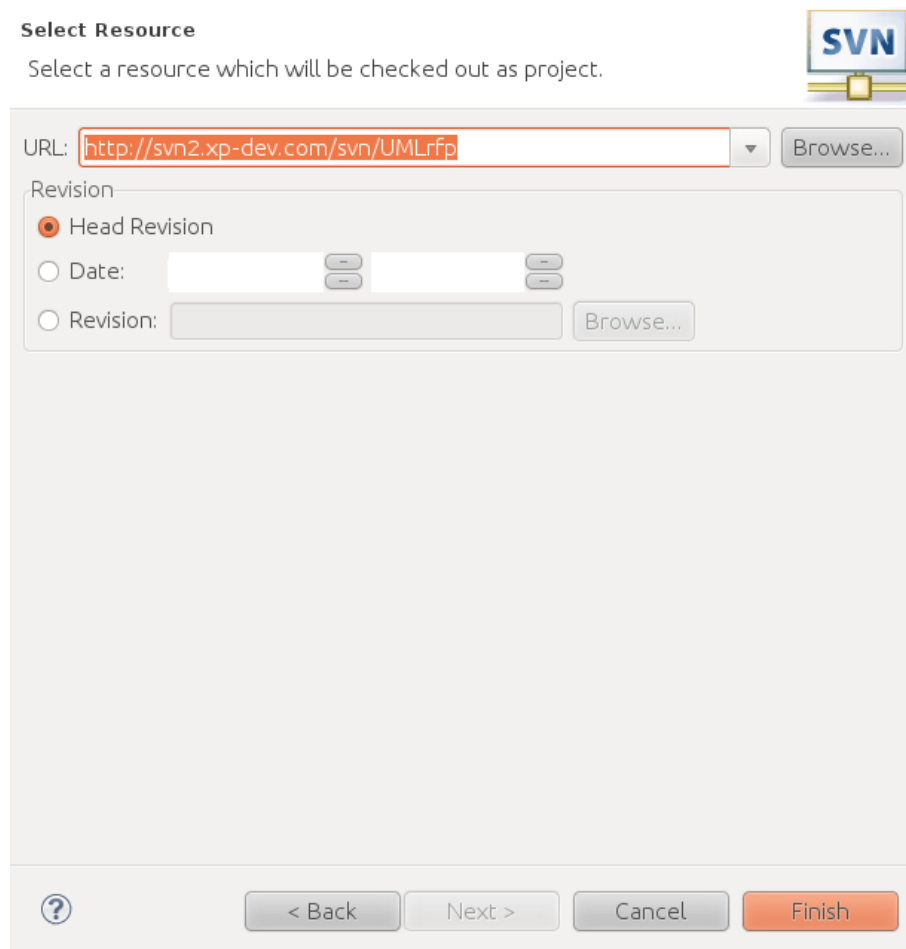


Figure A.5: Selecting the head revision

The next step is then to make Eclipse find the projects in the SVN repository by selecting the *Find projects in children of the selected resource* option in the wizard as shown in Figure A.6.



## A.2. Obtaining the code from an SVN repository

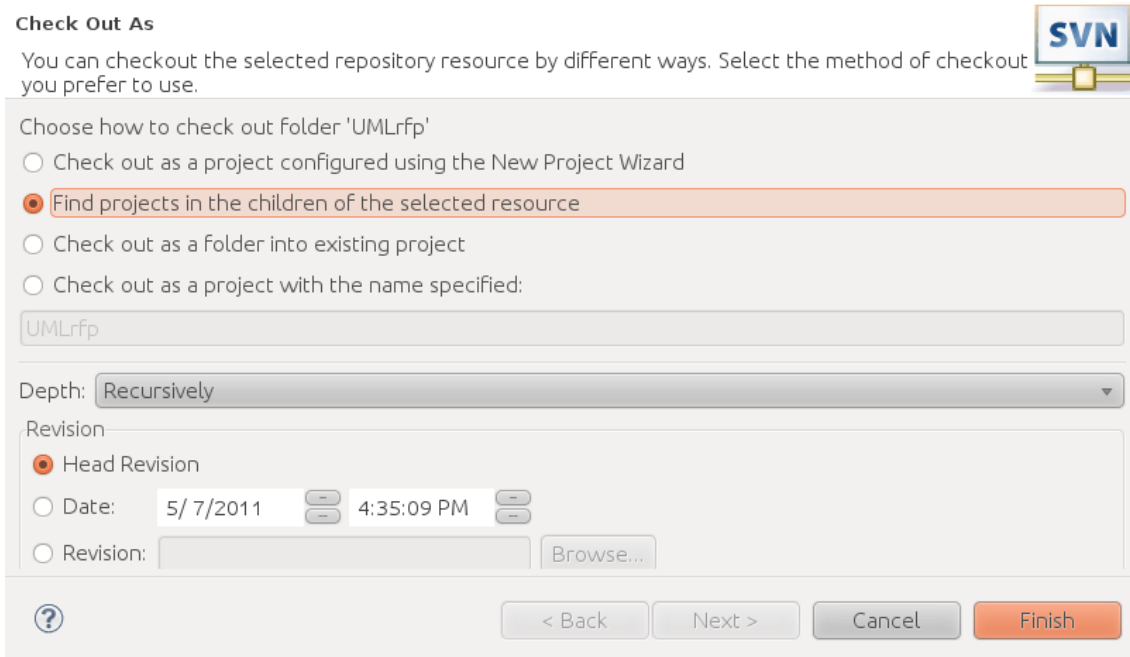
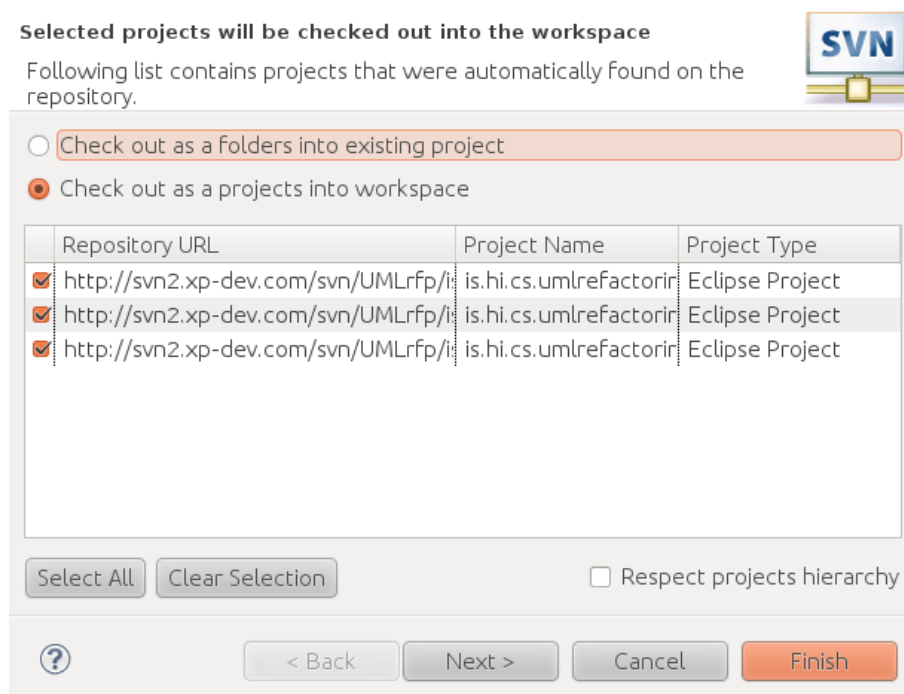


Figure A.6: Finding the projects in the SVN repository

The final step in the wizard is confirming that the correct projects have been found in the SVN repository as shown in Figure A.7, and instruct Eclipse to check the projects out as new projects in the workspace.



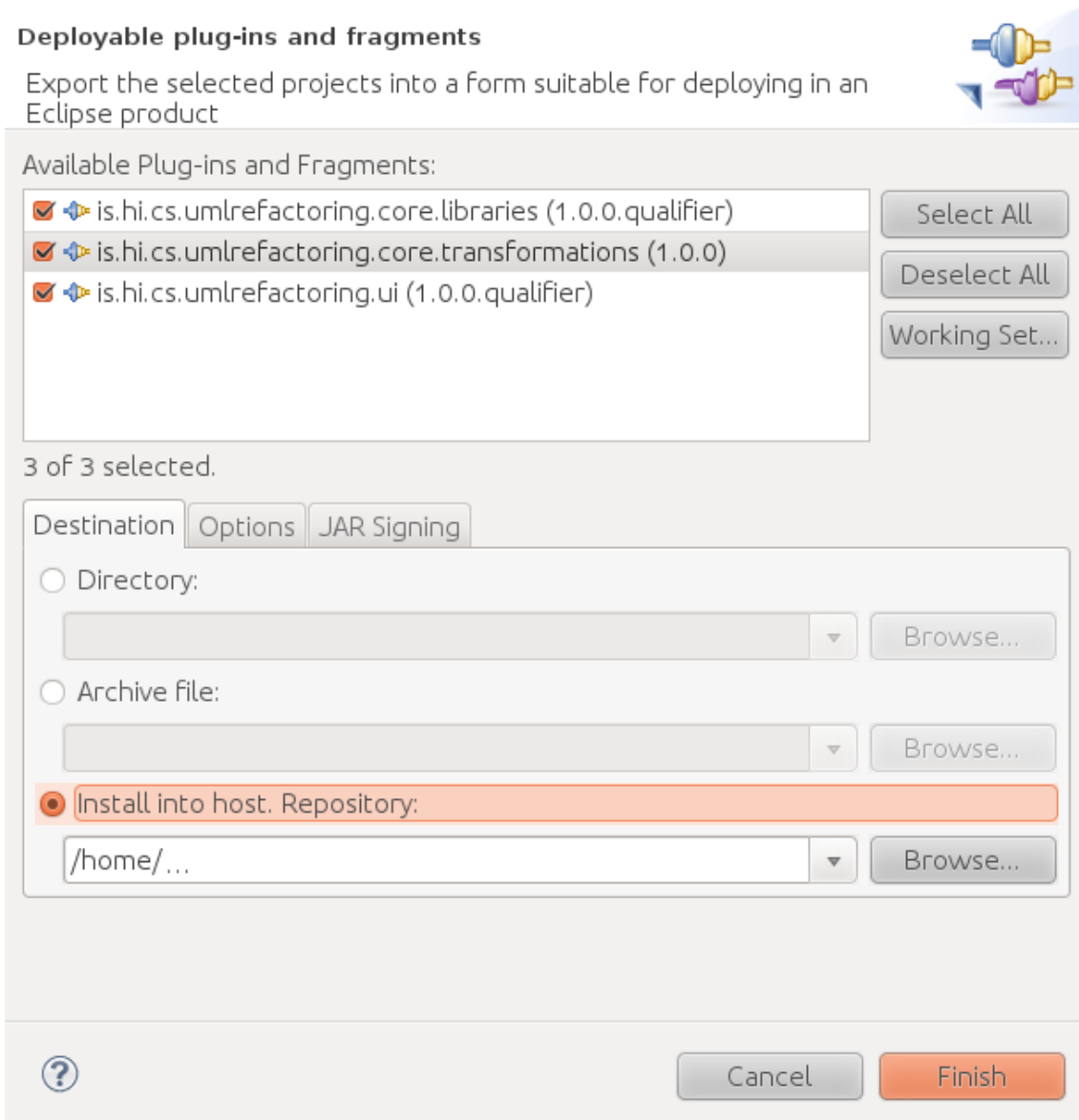
## A. The UML refactoring plugin

*Figure A.7: Confirming the projects found in the SVN repository*

The plugin is then ready to be installed as described in section A.3.

### A.3. Installing the plugin

To install the plugin the *is.hi.cs.umlrefactoring.ui* project is expanded in the Eclipse project explorer and the *plugin.xml* file opened. In the overview tab of *plugin.xml* the *Export wizard* is selected. In the first step of the wizard, the *is.hi.cs.umlrefactoring.ui*, *is.hi.cs.umlrefactoring.core.transformations* and *is.hi.cs.umlrefactoring.core.libraries* plugins are selected and the *Install into host* option selected as shown in Figure A.8. The *repository* field will be automatically filled by Eclipse.



*Figure A.8: Selecting the plugins for export*

The wizard concludes with calculating dependencies and installing the plugins into the host Eclipse instance.

## A.4. Using and debugging the plugin

When the code behind the plugin has been installed into Eclipse it will function as described in chapter 6 in the Papyrus editor. To view how the code is executed at

### A. The UML refactoring plugin

runtime, the plugin can be run in debug mode which will open a new Eclipse instance. To achieve this, one of the plugins should be selected in the *Project explorer* view and then the *Debug As -> Eclipse application* option selected in the *Run* menu as shown in Figure A.9.

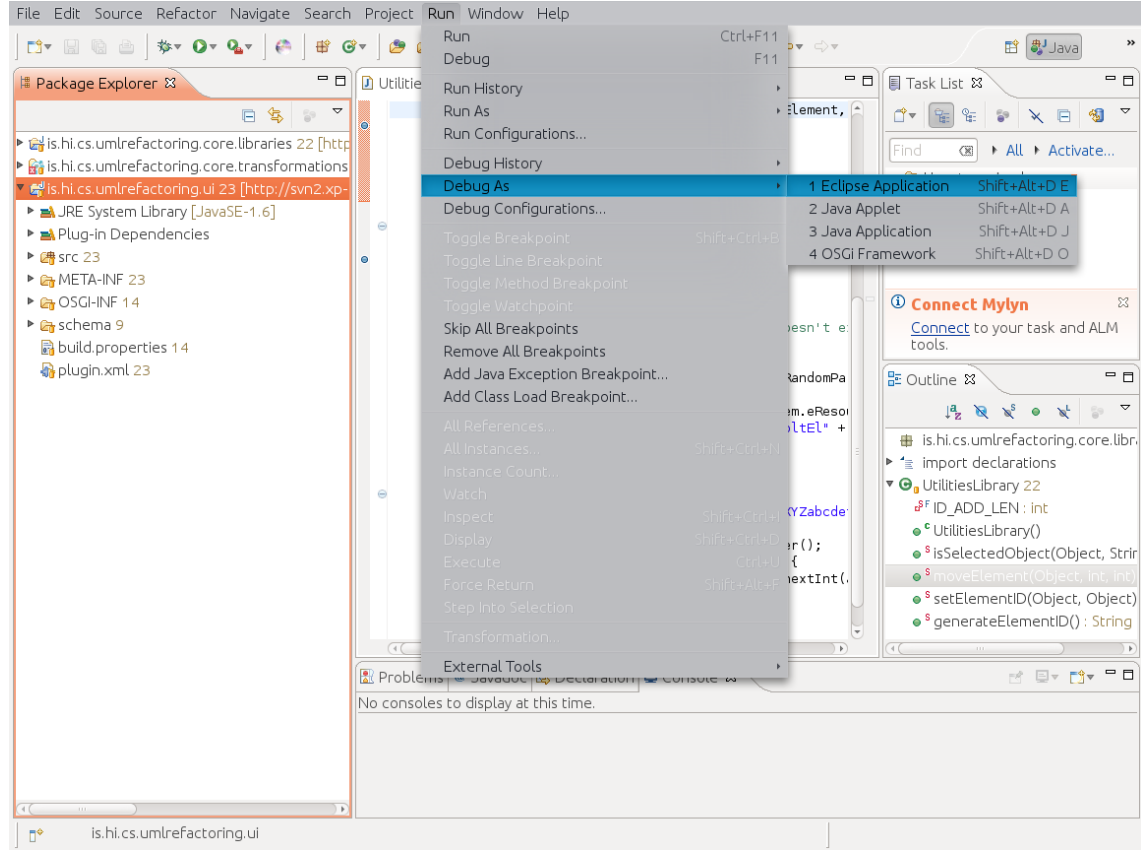


Figure A.9: The plugins debugged in a new eclipse instance

A new Eclipse install is started and breakpoints can be added in any Java code in the original instance to step through the code. However, to debug the QVTO transformation files, another method has to be used since the debugger is not activated for QVTO in another instance of Eclipse.

The first step to debugging a QVTO transformation file is to create a new debug configuration in Eclipse by selecting the *Debug Configuration* option in the *Run* menu. In the window that opens a new entry is added to the *Operational QVT interpreter entry* in the left menu. Then a new menu opens on the right as shown in Figure A.10 where the appropriate entries are input to each field. First the QVTO transformation is selected and then an optional trace file can be designated for examining the output of the transformation. Finally the Notation model and the UML model of the activity diagram to be transformed are selected.

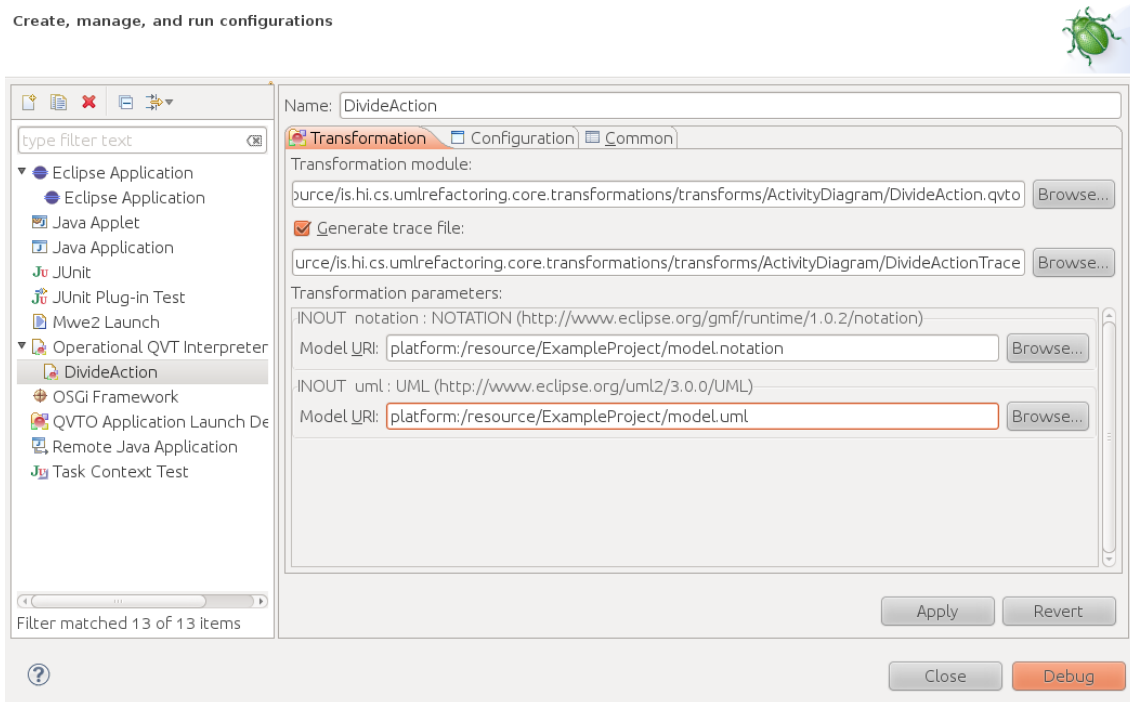


Figure A.10: Creating a debug configuration for a QVTO transformation

To complete the configuration, the *Configuration* tab is selected to add values to the configuration properties of the transformation. As described in chapter 5 the transformations need to receive the GUID of the Shape objects to be transformed. The Notation model file has to be opened with a text editor and a Shape object which points to a UML OpaqueAction found as shown in Figure A.11.

```
</children>
<children xmi:type="notation:Shape" xmi:id="Gv8A4Hl8EeCZa_NoNS-Opw" type="3007" fontName="Ubuntu" fontHeight="11">
  <eAnnotations xmi:type="ecore:EAnnotation" xmi:id="Gv8n8Hl8EeCZa_NoNS-Opw" source="ShadowFigure">
    <details xmi:type="ecore:EStringToStringMapEntry" xmi:id="Gv8n8Hl8EeCZa_NoNS-Opw" key="ShadowFigure_Value" value="<eAnnotations xmi:type="ecore:EAnnotation" xmi:id="Gv9PAHl8EeCZa_NoNS-Opw" source="displayNameLabelIcon">
      <details xmi:type="ecore:EStringToStringMapEntry" xmi:id="Gv9PAXl8EeCZa_NoNS-Opw" key="displayNameLabelIcon_value">
        <eAnnotations xmi:type="ecore:EAnnotation" xmi:id="Gv9PANl8EeCZa_NoNS-Opw" source="QualifiedName">
          <details xmi:type="ecore:EStringToStringMapEntry" xmi:id="Gv9PA3l8EeCZa_NoNS-Opw" key="QualifiedNameDepth" value="<children xmi:type="notation:DecorationNode" xmi:id="Gv9PBHl8EeCZa_NoNS-Opw" type="5003"/>
        <styles xmi:type="notation:HintedDiagramLinkStyle" xmi:id="Gv8A4Xl8EeCZa_NoNS-Opw"/>
        <element xmi:type="uml:OpaqueAction" href="model.uml#Gv49kHl8EeCZa_NoNS-Opw"/>
      </children>
    </children>
  </children>
</children>
```

Figure A.11: Finding a Shape object's GUID in a Notation model

The GUID of the Shape object is then copied into the value field of the configuration property as shown in Figure A.12.

## A. The UML refactoring plugin

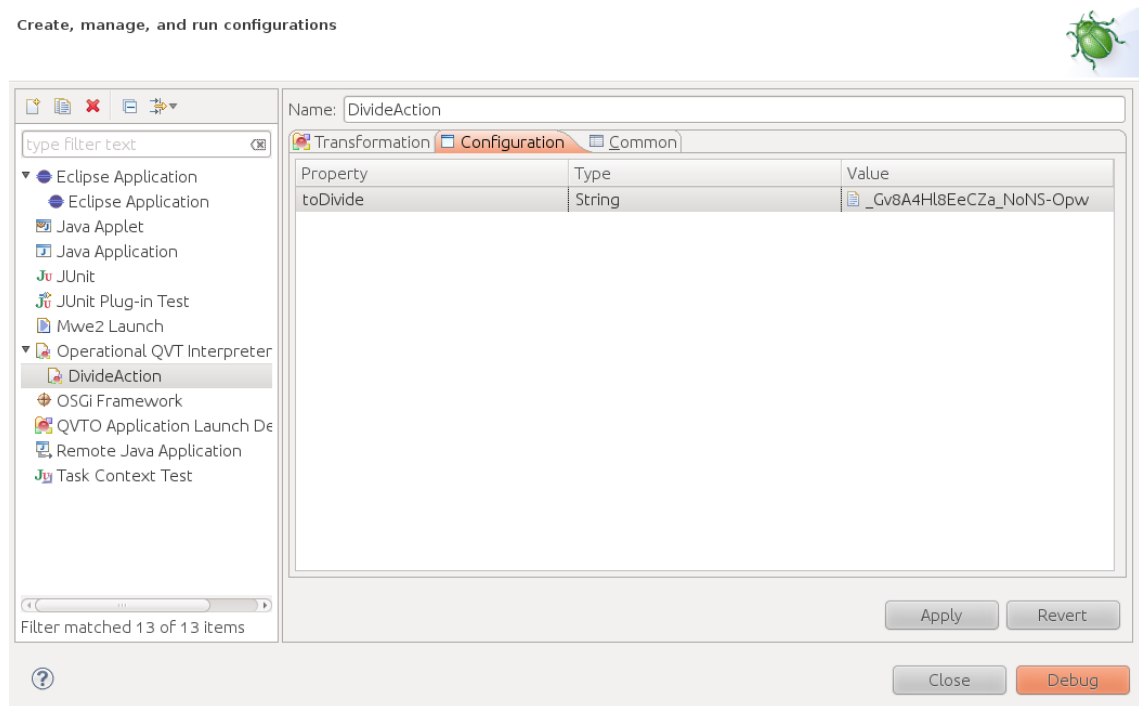


Figure A.12: Setting the value of a configuration property

Now the QVTO transformation can be stopped at breakpoints to step through the execution.