Refactoring Browser for UML

Marko Boger, Thorsten Sturm

Gentleware AG
Vogt-Kölln-Str. 30
22527 Hamburg, Germany
{Marko.Boger, Thorsten.Sturm}
@gentleware.de

Per Fragemann

Universität Hamburg, AG VSIS
Vogt-Kölln-Str. 30
22527 Hamburg, Germany
per@hamburg.de

Abstract

Refactoring is a corner stone in XP as well as other agile processes. Tools for an automatic support are beginning to appear, usually referred to as refactoring browser. Most of these are extensions to editors or IDEs and operate on code. This paper discusses how the idea of refactoring can be extended to UML models and presents a refactoring browser integrated in a UML modeling tool. Refactorings for the static architecture as well as dynamic behaviour models are presented.

Keywords

Refactoring, refactoring browser, development tool, UML

INTRODUCTION

Refactorings have gained wide attention, especially in the XP and agile process community. The idea was first formalized in the work of Opdyke [11] and Brant [5], made popular by Beck[2] and described in depth by Fowler [6]. Refactorings are techniques or recipes to improve the inner structure of software without changing its outward behaviour. XP and other agile processes propose to develop software in two iterative steps (or wearing two different hats). First, the desired behaviour should be implemented and secondly the structure of the code should be improved without changing any behaviour. This way, making changes later on becomes easier since the code structure becomes simpler and thus the developer is more agile.

Refactorings describe what can be changed, how this needs to be changed without altering the semantics, and what problems to look out for when doing so. A refactoring browser can help to automate the described steps and warn about possible conflicts.

Until now, refactorings have usually been discussed in the context of program code. All refactoring browser (to our knowledge) operate on code. Surprisingly though, refactorings themselves are often explained using UML notations. For us, this led to the question whether refactorings can directly be defined on the level of models rather than on code and whether refactoring browser could be implemented in the context of UML CASE tools rather than IDEs. This paper describes the outcome of this research work and discusses our findings.

In section 2 we discuss what refactorings make sense on the level of models and what additional refactorings can be found that make no sense on the code level but help on the model level. Section 3 describes how these refactorings were implemented in a tool. Section 4 gives an example how such refactorings and an according refactoring browser can be applied and section 5 rounds off with a conclusion.

REFACTORINGS FOR UML

Round trip engineering has reached a level of maturity that UML models and program code can be perceived as two different representations of the same thing, in the following simply called software. With such an environment in mind, the concept of refactoring can be generalized to something improving the structure of software instead of just its code representation.

For some refactorings it is natural to apply them on the code representation level. A refactoring like *extract method*, intended to separate code blocks from long method bodies into own methods, naturally applies to code. Others, like *rename class* or *move method up* have an identical effect whether they are applied to program code or a UML model. Others, like *replace inheritance by delegation* or *replace type key with state or strategy*, seem to fit better to a UML representation.

Defining the later kinds of refactorings on models and providing a refactoring browser for UML tools for these could be beneficial to more graphically oriented developers. But further more, since refactorings have only been thought of in the context of code, there might be new refactorings and benefits in tool support if applied to models. This paper focuses on such refactorings that apply to structure information of software that is not apparent from the code.

Conflict Detection

Detection of possible conflicts is a crucial part in the process of refactoring. Martin Fowler encourages the use of unit tests to prevent unwanted side effects from slipping into working code. But he leaves it up to the user to find out which side effects might be present in a specific situation.

In our work we focus on the automatic detection of conflicts. Each refactoring may introduce lots of errors into code if applied incorrectly. Therefore each of the proposed refactorings discussed below has been closely examined to determine likely conflicts caused by its potential use.

Conflicts are divided into warnings and errors. Warnings

indicate that a refactoring might cause a side effect, while leaving the model in a well-formed state. For example, renaming a method so that it overrides a superclass's method may be behaviour-preserving in some cases, but a major unwanted design change in others. The code will in either case remain compilable. Errors on the other hand indicate that an operation will cause damage to the code or model.

Some UML-refactorings are very likely to produce conflict warnings, even if the intended refactoring will not alter the models behaviour. Others will mainly report major model-changes. We have come to the conclusion that while all conflicts have to be presented to the user, he must be able to override the refactoring browsers conflict warnings. This enables him to perform several refactorings in a row, where single ones are not behaviour-preserving but the sum of them is, or to correct conflicts by other means.

Static Structure Refactorings
UML class diagrams are used to design and visualize the static architecture of a software system. It is clear that some refactorings that are known from code-oriented refactoring can directly be applied to class diagrams. But while code is a linear representation and may not display the dependencies and structure well, UML is a two dimensional and graphical representation. Improvements to the architectural structure may be simpler to spot in a class diagram than a code editor. Also, after a possible refactoring is identified, consequences of its application may be better overviewed in UML.

Due to the mesh of dependencies of object-oriented software structures, many refactorings like renaming, deletion, moving of methods, classes or attributes have more effects than just the local change. Often such problems are caused by inheritance and polymorphism. Renaming a method, for example can have the effect that a method higher in the inheritance hierarchy now is overridden (by the new name) or not overridden anymore (by the old name).

Refactorings that change the structure, like the replacement of inheritance by delegation or the extraction of a common interface from a set of classes are more apparent on a model level.

But it is important to note that UML consists of more than just the class diagram. Class diagrams are probably the most important and certainly the most used part of UML but there is more to it. Most refactorings described in the literature so far apply to the static structure and their effects would become apparent in the class diagram only. However, the class diagram is not able to express the dynamic behaviour nor a business process or business requirements. Also code is not the optimal choice to express these aspects of software.

UML offers mechanisms to express these with the help of several different diagram types. The view that software not only consists of code but that these aspects belong to software as well is spreading. We want to restrict ourselves to activity and state diagrams next to class diagrams in this paper.

State Machine Refactorings
State diagrams are often used to express the protocol of how the set of operations for a class should be used. That is, it defines in what order operations should be called and what order of calls are not allowed. Refactoring such a protocol means changing the way the protocol is presented without changing the protocol itself.

The following are a selection of refactorings on state diagrams that we were able to identify as well as implement:

- *Merge States* is used to form a set of states into a single one. Inner transitions between the states are removed, external transitions are redirected to the new state, internal transitions and actions are moved into the new state. Warnings are generated if (among others) more than one of the selected states has got transitions out of the group, or if the only entry-action found was not located in the "virtual initial state" of the selected group.
- *Decompose Sequential Composite State* moves the contained elements of a composite state out of it and removes it afterwards. Entry- and exit-actions are moved to the appropriate states, transactions leaving or reaching the composite state are redirected or copied if necessary. Warnings have to be issued for example if entry- and exit-actions are detected, or if a do-activity will be removed. This refactoring is mainly used in conjunction with *form composite state*.
- *Form Composite State* creates a new composite state and moves the selected states into it. Common transitions are extracted to the border of the new state, and appropriate default and completion states are linked to the initial state and the completion transition. Warnings have to be generated if no default state can be found, or if more than one candidate for default state or completion state are equally suitable.
- *Sequentialize Concurrent Composite State* creates the product automaton for a concurrent state and removes the contained elements from the concurrent state. It is mainly used in conjunction with merge states to simplify complex models while preserving behaviour. This refactoring is only applicable to wellformed concurrent states, therefore it checks for initial states if a region is not targeted by a fork directly, for example.

Activity Graph Refactorings
Activity diagrams are often used to describe the business process. They can also be used to describe algorithms in a more graphical way. But while the later is usually better expressed in code, there is no adequate way to express the business process in code. The value of a clear understanding of this process is undoubted and activity diagrams are a good means to model them. Refactorings of

these do not change the process but only the way it is represented in the model. Here is the selection of implemented refactorings:

- *Make Actions Concurrent* creates a fork and a join pseudostate, and moves several sequential groups of actions between them, thus enabling their concurrent execution. The refactoring detects whether the modification leads to a wellformed model by checking for errors like transitions between two groups, or states having no transition-path to the last state of a group. Warnings have to be issued if one group writes to a variable that is accessed by another group.
- *Sequentialize Concurrent Actions* removes a pair of fork and join pseudo states, and links the enclosed groups of action states to another. Warnings are generated if commonly used variables are found (a real concurrent algorithm cannot simply be sequentialized), and the detection suite also checks for non-wellformed transitions.

A REFACTORING BROWSER FOR UML

The refactorings described above have been implemented in a refactoring browser for UML. Some of the implementation aspects and a description on the developed interface are described in this section.

UML is more than a graphical notation and UML tools are (usually) more than just specialized drawing tools. While code editors are usually simply advanced text editors, UML editors already have a semantically rich internal representation, a repository, which (for some tools at least) is even based on a standardized UML meta model [10]. Refactorings on UML models can exploit and operate on this meta model structure. This makes the implementation of a UML refactoring browser more straight forward than a code based one for which a predefined meta structure may not exist.

The implementation described here is done as part of the Gentleware tool Poseidon for UML [9], which originates from the open source project ArgoUML [1]. Its repository is directly generated from the UML 1.3 meta model [10]. Refactorings are implemented as controllers on this model.

The user interface of most code based refactoring browsers are simply implemented as context menus. We have chosen differently and implemented it as a pane that can be seen at the same time as the diagram and the navigation pane. It consists of three compartments.

In the first compartment refactorings are proposed based on the current selection. If, for example, a method is selected, it proposes the refactoring *rename method*. If at the same time a superclass is also selected, the proposer suggests the refactoring *move up method*. Selecting a refactoring displays a short description of the refactoring and its effects as well as a dialog for entering parameters (like the new name or application to derived methods) are displayed. The third section shows possible conflicts that can occur if the refactoring is executed. It also holds the button to execute a refactoring as well as (in the future) to undo it again.

AN EXAMPLE

Some small examples should show how refactorings for a UML model can help to improve a design and how the usage of a refactoring browser helps to avoid refactoring pitfalls by addressing possible conflicts. The overall scenario is taken from the default example that is shipped with Gentleware's Poseidon for UML. An imaginary company called Softsale is selling digital products over the internet. The handling of orders and customer relations are modeled by the sample UML model. For this context, the example is limited to modeling the verification of a new customer.

Part of the UML model is the modeling of the customer itself. There are two classes, a `Customer` that has two associations to a `DeliveryAddress`. The role names of these associations are `deliveryAddress` and `invoiceAddress` (see Fig. 1).

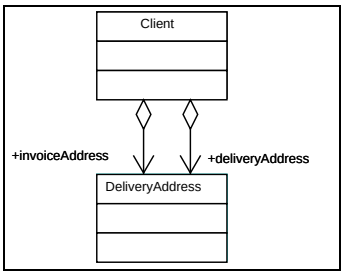

Fig. 1 Class diagram before *rename class*.

Having multiple associations to the class `Customer` with one role name equal to the class name itself, it looks like the name for the class `DeliveryAddress` is not the perfect one. Using the refactoring *rename class*, the name of the class will be changed to `Address` to better reflect its usage (see Fig. 2). For this is a rather simple refactoring, no conflicts are expected to occur.

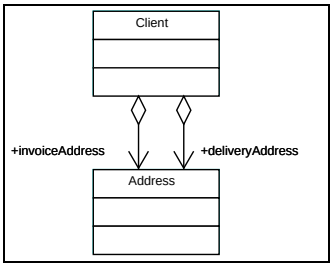

Fig. 2 Class diagram after *rename class*.

The verification of a new customer's delivery address is started by requesting the appropriate address and validation of the received information. Therefore, the address starts in the state `not verified`, goes through the verification process (`requested`, `received`) and either returns to `not verified` or moves to `veri-`

fied, depending of the result of the verification (see Fig. 3).

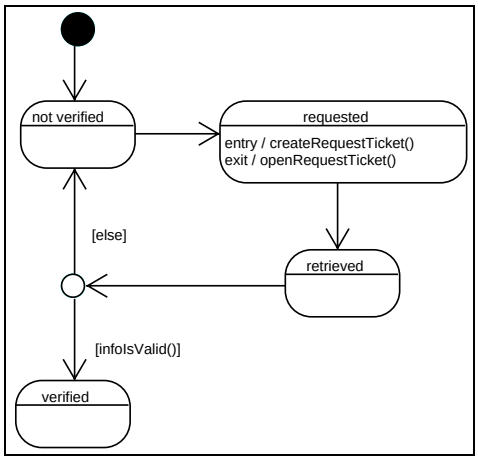

Fig. 3 State diagram before *merge state*.

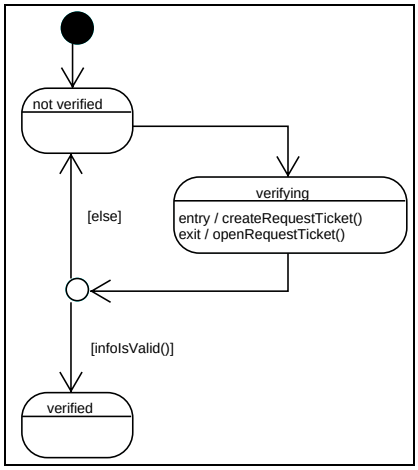

Fig. 4 State diagram after *merge state*.

The states `requested` and `received` are sequential and merging them would not change the logic of the process. The refactoring *merge states* merges them to a new state named `requested,received`. To better reflect its purpose, we changed the name to `verifying` in a second step without using the refactoring browser (see Fig. 4).

Fig. 5 Warnings as presented by the tool.

The state requested has an entry action as well as an exit action. Having an exit action here causes a conflict in the refactoring, that is displayed in the refactoring browser along with a short explanation. Now it's up to the user to either rethink what he wants to do or to execute the refactoring and solve the conflict afterwards. In this case, we ignore the conflict, because the exit action can be used for the merged state as well.

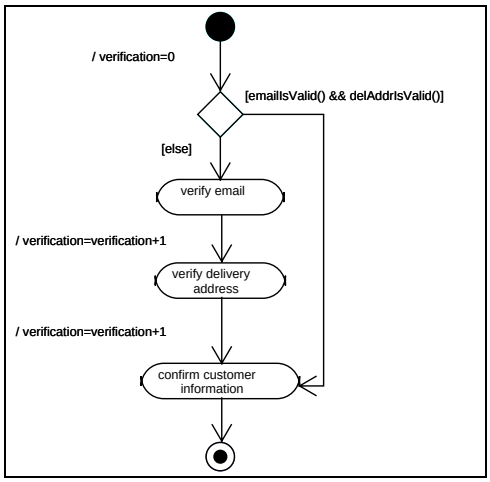

Fig. 6 Activity diagram before *make actions concurrent*.

The complete process of the verification of a new customer is expressed in an activity diagram. The verification starts with verifying the email address, followed by verifying the delivery address (see Fig. 6).

Because both parts of the verification process have no dependencies upon each other, the refactoring *make actions concurrent* can be used to improve the model (see Fig. 7).

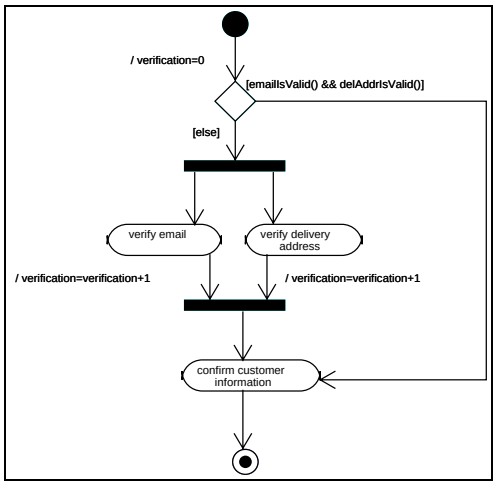

Fig. 7 Activity diagram after *make actions concurrent*.

Because the action states `verfify email` and `verify delivery address` both use a variable called `verification`, the refactoring browser detects a conflict here. Because the variable `verification` is used to increment it, we can ignore the conflict here and continue the refactoring.

Fig. 8 Warnings as presented by the tool.

The example shows, that there is room for improvement by using refactorings not only in the static structure of the model, but also in state and activity diagrams.

CONCLUSION

Refactoring as the disciplined process of improving the structure of program code without changing its behaviour can be generalized to also be applied to UML models. Many known refactorings can directly be transferred and implemented in CASE tools. Finding possible refactorings for the static structure may be simpler in a two dimensional graphical representation. In addition new refactorings for activity graphs or state machines can be found.

The described work is fully implemented in a demonstratable prototype. Details are described in [8]. It will be further developed to become a plug-in for the UML CASE-tool Poseidon for UML which is distributed by Gentleware.

REFERENCES

1. ArgoUML. ArgoUML, Object-oriented design tool with cognitive support, Open Source Project, at http://www.argouml.org.
2. Beck, Kent. Extreme Programming explained: Embracing Change. Reading, Mass., Addison-Wesley, 1999.
3. Boger, Marko et al. Extreme Modeling, in Succi, G and Marchesi, M. Extreme Programming Examined, p.175, Addison-Wesley, 2000.
4. Boger, Marko and Sturm, Thorsten. Tools-support for Model-Driven Software Engineering, in Evans, A. et al. Proceedigs of Practical UML-Based Rigorous Development Methods. Workshop at <<UML>>2001 conference. Gesellschaft für Informatik, 2001.
5. Brant, John and Roberts, Don. Refactoring Browser, at http://st-www.cs.uiuc.edu/~brant/RefactoringBrowser.
6. Fowler, Martin. Refactoring, Improving The Design of Existing Code, 1999.
7. Fowler, Martin. Refactoring Home Page, at www.refactoring.org.
8. Fragemann, Per. Refactoring von UML-Modellen. Diploma Thesis (to be published). University of Hamburg, Germany, 2002.
9. Gentleware AG. Poseidon for UML, a UML CASE tool based on ArgoUML, at http://www.gentleware.com.
10. Object Management Group. Unified Modeling Language, Version 1.3, at http://www.omg.org.
11. Opdyke, William F. Refactoring Object-Oriented Frameworks. *Dissertation*, University of Illinois at Urbana-Champaign, 1992.