Refactoring Browser for UML

Marko Boger, Thorsten Sturm, Per Fragemann

Gentleware AG, Vogt-Kölln-Str. 30, 22527 Hamburg
{boger,sturm,fragemann}@gentleware.de

Abstract. Refactoring is a corner stone in a number of agile processes like Extreme Programming (XP). Tools for an automatic support are beginning to appear, usually referred to as refactoring browsers. 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 modelling tool. Refactorings for the static architecture as well as for the dynamic behaviour are presented.

1 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 [4], made popular by Beck [2] and described in depth by Fowler [8]. 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 second, 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 the modification can be done 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 browsers (to our knowledge) operate on code. Surprisingly though, a number of refactorings themselves are explained using UML notations (like in [8]). 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 browsers 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 results.

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 of how such refactorings and

an according refactoring browser can be applied and section 5 rounds off with a conclusion.

2 Refactorings for UML

Roundtrip engineering has reached a level of maturity that UML models and program code can be perceived as two different representations of the same artifact, 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 Up Method*, have an identical effect whether they are applied to program code or a UML model. Others, like *Extract Interface*, 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.

2.1 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 of these side effects might be acceptable in the specific situation.

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

Preconditions are defined for every single refactoring and mapped to appropriate messages. The currently selected model elements are tested to meet the preconditions for the selected refactoring. This way a list of broken preconditions is created for the refactoring and the corresponding messages are presented to the user as conflicts.

Conflicts are grouped 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 superclasses method may be behaviour-preserving in some cases, but a major unwanted design change in others. The resulting code will in either case remain compilable. Errors, on the other hand, indicate that a operation will cause damage to the model and break its consistency.

Some UML refactorings are very likely to detect conflicts, even if the intended refactoring will not alter the model's 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 browser's suggestions. This enables him to perfom several refactorings in a row, with the last one restoring the model's behaviour.

2.2 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 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 in a code editor. Also, after a possible refactoring is identified, consequences of its application may be overviewed better in UML.

Due to the mesh of dependencies in 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 deeper in the inheritance hierarchy is now 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, neither a business process or business requirements. Using the power of UML to express the dynamic behaviour in diagrams gives a better overview especially of complex behavioral aspects, focussing just on the important part of the software. E.g., while having to trace the call sequence of a given scenario might cause switching between a number of classes in the code editor, the complete scenario can be expressed in one sequence or collaboration diagram.

The view that these aspects belong to software is spreading. In combination with mature roundtrip engineering features, using models for refactoring software combines the power of agile processes with the expressiveness of UML. Although refactorings are possible for all types of diagrams defined in UML, we want to restrict ourselves to activity diagrams and state charts next to class diagrams in this paper.

2.3 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 orders of calls are not allowed. Refactoring such a protocol means changing the way the protocol is presented without changing the protocol itself.

- *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 (among others) generated if more than one of the selected states has transitions to other states outside 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* helps moving the contained elements to the composite states level and removing 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 if exit- 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 well-formed concurrent states, therefore, it checks for initial states if, e.g., a region is not targeted by a fork directly.

2.4 Activity Graph Refactorings

Activity graphs describe ways of running through processes. They are focussed on the stepwise execution of activities or the flow of objects. Refactoring the graph means changing the execution order of activities or the way objects are moving within a process without changing the overall result of it.

- *Make Actions Concurrent* creates a fork and a join pseudostate, and moves several sequential groups of actions in between, thus enabling their concurrent execution. The refactoring detects whether the modification leads to a well-formed model by checking for errors like transitions between two groups,

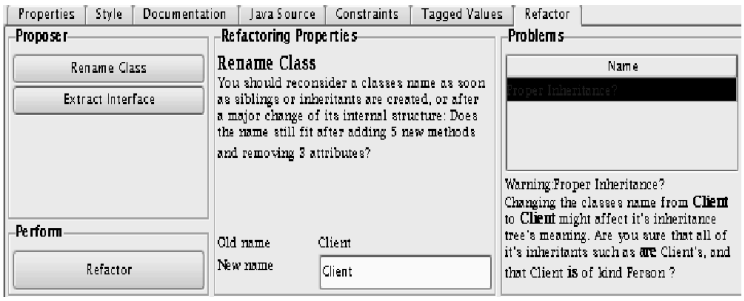


Fig. 1. User Interface of the Refactoring Browser

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 pseudostates, and links the enclosed groups of action states to another. Warnings are generated if commonly used variables are found (a concurrent algorithm cannot be sequentialized), and the detection suite also checks for malformed transitions.

3 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 [1], which originates from the open source project ArgoUML [5]. Its repository is directly generated from the UML 1.3 meta model [10]. Refactorings are implemented as controllers on this model. The tool implements all the refactorings described for state machines and activity graphs. Static structure refactorings are implemented down to the level of method signatures. Refactorings covering the method bodies are not implemented due to missing notation for them in UML.

The user interface of most code based refactoring browsers is 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 deduced methods). The third section shows possible conflicts or problems 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.

For the static structure refactorings the challenging part was to check for possible conflicts. These are only displayed if they actually apply, including the concrete source of problem. Conflicts can be warnings and errors. Errors determine concrete problems that will break the consistency of the model when the refactoring is executed. Warnings give hints on possible problems and occuring side-effects, e.g., renaming of a method in a superclass can stop it from being overwritten in a subclass. Refactorings with errors should not be executed. However, the user can choose to overrule the warnings and errors and execute the refactoring anyway. This can be useful when following refactorings restore the consistency of the model or the designated behaviour.

4 An Example

Some small examples should show how refactorings for a UML model can help improving a design and how the usage of a refactoring browser helps avoiding refactoring pitfalls by addressing possible conflicts. The overall scenario is taken from the default example that is shipped with Gentleware's Poseidon for UML.

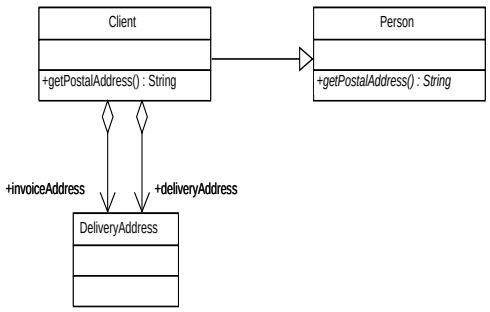


Fig. 2. Client Structure

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. It shows some potential refactorings not only for the structure but also for the behaviour of the modeled software.

4.1 Static Structure Refactoring

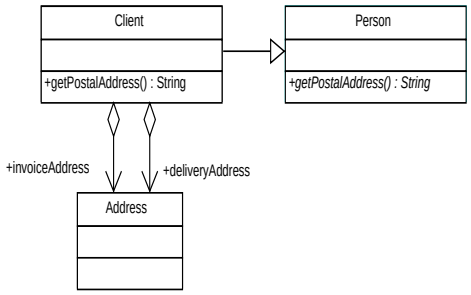


Fig. 3. Client Structure after Refactoring

Part of the UML model is the modeling of the customer itself. There are three classes: a `Person` that has an abstract operation returning a string representation of the person's postal address and a `Customer` that has two associations to a `DeliveryAddress`. The names of the ends of these associations are `deliveryAddress` and `invoiceAddress` (see Fig. 2).

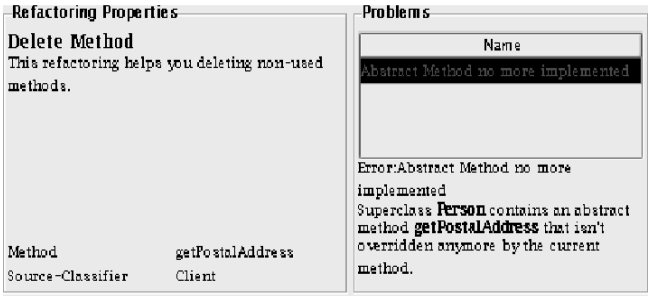


Fig. 4. Errors in Client Structure Refactoring

Having multiple associations to the class `Customer` with one association end 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. 3). As this is a rather simple refactoring, no conflicts are expected to occur.

Deleting the operation `getPostalAddress` in the class `Customer` causes an error. The class would no longer override the abstract operation `Person.getPostal-Address`. Please note that the errors are presented in red while warnings are presented in blue. A click on the short name of the conflict (either error or warning) shows a more detailed description of the reasons for raising the it (see Fig. 4).

4.2 State Machine Refactoring

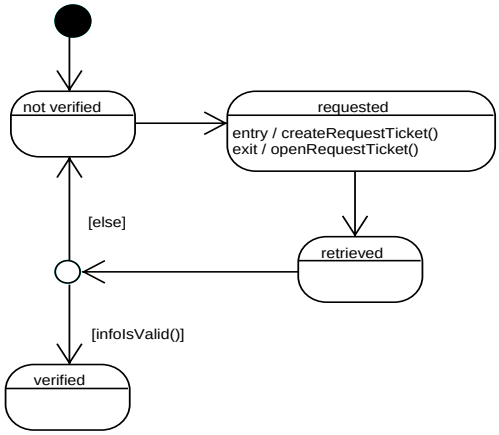


Fig. 5. Address Verification Activity

The verification of a new customer's delivery address is started by requesting the appropriate address and validating the received information. Therefore, the address starts in the state `not verified`, steps through the verification process (`requested`, `retrieved`) and either returns to `not verified` or moves to `verified`, depending of the result of the verification (see Fig. 5).

The states `requested` and `retrieved` 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,retrievedverifying`. To better reflect its purpose, we changed the name to `verifying` in a second step without using the refactoring browser (see Fig. 6).

The state requested has an entry action as well as an exit action. Having an exit action here causes a conflict in the refactoring. It is displayed in the refactoring browser along with a short explanation. The exit action might not be appropriate for the merged state, because it is not the exit action of the last state in the selected sequence. Now its up to the user to either rethink what

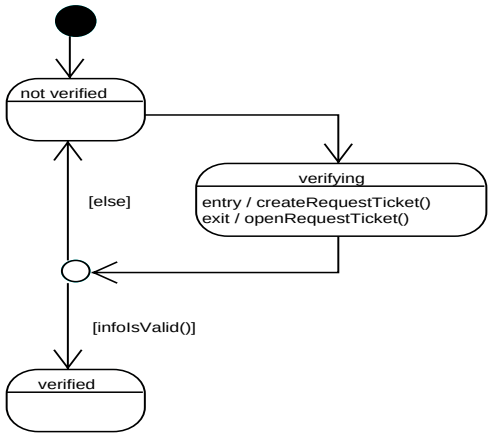


Fig. 6. Address Verification after Refactoring

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.

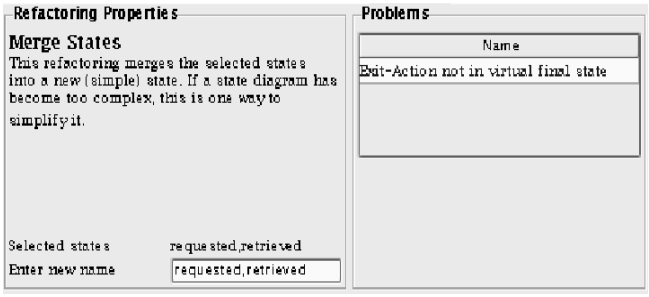


Fig. 7. Conflicts in Address Verification Refactoring

4.3 Activity Graph Refactoring

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. 8).

The verified data is completely independent and the defined sequence might cause a delayed verification of the delivery address. 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. 9).

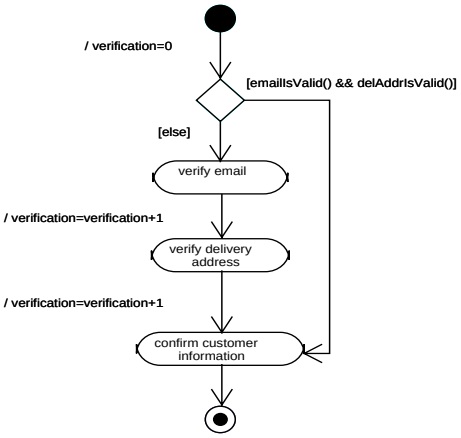


Fig. 8. Client Verification Activity

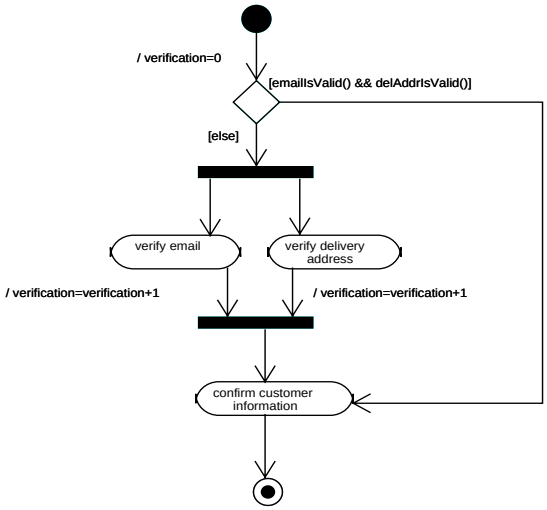


Fig. 9. Client Verification after Refactoring

Because the action states `verify email` and `verify delivery address` both use a variable called `verification`, the refactoring browser detects a conflict here. The value of the variable might be used to in one of the concurrent action state while another one is changing it. Because the variable verification is incremented only, we can ignore the conflict here and continue the refactoring.

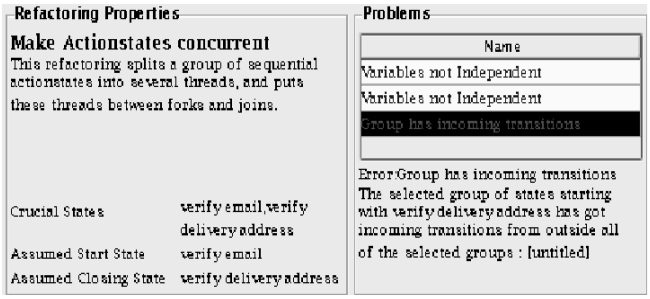


Fig. 10. Conflicts in Client Verification Refactoring

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

5 Future Work

Although we find our tool highly valuable and start to use it in our own day-to-day modelling work, there are currently some limitations. The described refactorings change the model and affect the generated code. We use it in combination with elaborated code generators which have been described in [6] and [3]. This helps increasing the degree of automation in all development phases focussed on models or code.

However, as long as modeling of method bodies is not covered by the UML specification, purely model-based refactoring approaches will not provide the full power of refactorings as described in [8]. Therefore, we plan to combine our refactoring tool for models with a refactoring tool operating on code. Any refactoring will then be reflected in the model as well as in those parts of the code not covered by the model.

The definition of errors and warnings for a single refactoring is a challenging task. Currently, they are defined following the UML standardization and some best practices. The development of some sort of methodology or cookbook would leverage their maturity and usefulness.

The usage of meta models even for code is constantly evolving. Recent and upcoming standardizations try to fill the current gap between them (e.g the Java

Metadata Interface[12] and the UML 2.0 standardization [10]). Future refactoring tools will reflect this by melting UML-based and code-based refactorings into model-based refactorings to overcome the mentioned limitations.

One of the tenets of refactoring is repeated testing to ensure that a refactoring did indeed not break any implemented behaviour. Certainly this is more difficult on models. However, models can be taken to a point where they become executable and thus testable. Especially state diagrams can effectively be transformed to code. This not only helps to visualize a running system but also improves automated testing (test for reached states) and documentation.

6 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. Combining refactoring on the level of UML models with roundtrip engineering of them allows for using UML in agile development processes.

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

7 Acknowledgement

This work was developed in a cooperation between the University of Hamburg and Gentleware, and we would like to thank both parties. We would also like to thank the community around the ArgoUML open source project.

8 References

References

[1] Gentleware AG. Poseidon for UML. *http://www.gentleware.com.*

[2] Kent Beck. *Extreme Programming explained: Embracing Change.* Addison-Wesley, 1999.

[3] Marko Boger and Thorsten Sturm. Tools-Support for Model-Driven Software Engineering. In A. Evans, editor, *Proceedings of Practical UML-Based Rigorous Development Methods. Workshop at <<UML>>2001 conference.* Gesellschaft für Informatik, 2001.

[4] John Brant and Don Roberts. Refactoring Browser. *http://st-www.cs.uiuc.edu/~brant/RefactoringBrowser.html.*

[5] ArgoUML Community. ArgoUML, Object-Oriented Design Tool with Cognitive Support.
http://www.argouml.org.

[6] Marko Boger et al. Extreme Modeling. In G. Succi and M. Marchesi, editors, *Extreme Programming Examined*, page 175ff. Addison-Wesley, 2000.

[7] Martin Fowler. Refactoring Home Page.
http://www.refactoring.org.

[8] Martin Fowler. *Refactoring - Improving the Design of Existing Code.* Addison-Wesley, 2000.

[9] Per Fragemann. Refactoring UML. Master's thesis, University of Hamburg, AG VSIS, 2002.

[10] Object Management Group. Unified Modeling Language.
http://www.omg.org/uml.

[11] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[12] Java Community Process. Java Metadata Interface.
http://www.jcp.org/jsr/detail/40.jsp.