Refactorings without Names

Friedrich Steimann
Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
58084 Hagen
Germany
steimann@acm.org

Jens von Pilgrim
Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
58084 Hagen
Germany
Jens.vonPilgrim@feu.de

ABSTRACT
As with design patterns before, the naming and cataloging of refactorings has contributed significantly to the recognition of the discipline. However, in practice concrete refactoring needs may deviate from what has been distilled as a named refactoring, and mapping these needs to a series of such refactorings — if at all possible — can be difficult. To address this, we propose a framework of specifying refactorings in an ad hoc fashion, and demonstrate its feasibility by presenting an implementation. Evaluation is done by simulating application through a user on a set of given sample programs. Results suggest that our proposal of ad hoc refactoring is, for the investigated scenarios at least, viable.

Categories and Subject Descriptors
D.2.6 [Software Engineering]: Programming Environments;
D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—Restructuring, reverse engineering, and reengineering

General Terms
Design, Languages, Theory

Keywords
Refactoring, program restructuring

Few programmers would tolerate if one of these tools silently changed the behaviour of their programs, and we consider a refactoring tool which does so as flawed. With our work, we strive for a level of correctness beyond what “a good test suite” can ascertain.

1. INTRODUCTION
In his seminal book [1], Fowler compiled a list of 72 named refactorings that have since, like design patterns before, made it into the active vocabulary of software developers. However, as is often the case with standardized solutions, in practice slight variations of the problem to be solved suffice to prevent users from using them. In such cases, individualized solutions are needed.

In this paper, we sketch the idea of ad hoc refactoring — or refactorings without names — as the answer to one-of-a-kind refactoring problems, and present a first functioning prototype of a refactoring tool as a proof of concept. Our implementation utilizes state-of-the-art constraint-based refactoring techniques as described, e.g., in [6, 5], allowing us to reuse much of the apparatus that has been conceived previously for specifying named refactorings. Our evaluation, which is somewhat limited in its scope, reflects the current restrictions of constraint-based refactoring; we have left pushing the boundaries of what can be done further, to future work.

2. A REFACTORING VISION
Imagine a programmer wanting to perform a refactoring that (s)he conceives of as a number of elementary changes — not necessarily themselves refactorings — that together lead to the targeted design. Imagine that, to carry out the refactoring, the programmer presses an IDE button labelled “begin refactoring”, performs the intended changes in the IDE’s editor, and finally presses “end refactoring”. Imagine that the refactoring engine then
1. checks whether the performed changes constitute a complete refactoring;
2. if not, checks whether there are additional changes that would make it one and, if so, performs these changes;
3. if not, rolls back all performed changes and reports to the user.

Because each so performed refactoring pursues its own purpose, we refer to our refactoring vision as ad hoc refactoring.

The following concrete application scenarios illustrate why ad hoc refactoring would indeed be desirable.

APPLICATION SCENARIO 1
Consider a programmer wanting to transform some classes written for a specific application into a library to be shared with others. In order to enable reuse of the
classes, the accessibility of various members has to be increased from package or protected to protected or public. In Java, this may change the meaning of the program, since generally, accessibility has an effect on overriding, shadowing, and overloading resolution (and thus binding) [2, 3]. In our vision of ad hoc refactoring, the programmer presses “begin refactoring”, replaces the access modifier(s) as intended, and presses “end refactoring”. Should it note a change of binding, the refactoring engine will either perform — user permitting — additional changes necessary to restore the binding (e.g., renaming or introducing qualifiers) or roll back the change(s); otherwise, it will do nothing.

APPLICATION SCENARIO 2 Consider a class with two fields code and id whose names have been confused so that they need to be swapped. Doing this as two consecutive RENAME FIELD refactorings is impossible, since the first RENAME FIELD would leave the program in an inconsistent state from which the second cannot recover. Instead, three RENAME refactorings will be necessary (the first renaming one field to a temporary name, the second renaming the other field as intended, and the last replacing the temporary name with the other’s original name), which is awkward. In fact, it is so awkward that one might be tempted to conceive of the name swap as a new standard refactoring, named SWAP FIELD NAMES, that, from a user perspective, does it all in one step. However, introducing new refactorings as needed leads to a proliferation of refactorings, and will eventually swamp the IDE and its users. With our ad hoc refactoring tool at hand, the programmer need not care whether a refactoring has been thought, defined, or implemented before — (s)he just renames the fields in the editor and trusts the refactoring tool to perform all necessary checks and updates.

APPLICATION SCENARIO 3 Consider the case that a class hierarchy is to be persisted using an existing OR-mapper which maps each class to a separate table, but only if the class adds fields to those inherited from its superclass. In order to keep the number of tables small, the programmer decides to pull up fields to superclasses, renaming each field to indicate its origin (e.g., a field salary of a class Employee is renamed to “employeeSalary” after it has been moved to the superclass Person). Rather than performing a PULL UP FIELD refactoring and a RENAME refactoring separately on every field in question, (s)he would just move and rename all fields in the editor in one sweep, and would be done.

Many variations of these scenarios can be envisioned. For instance, a change of naming conventions may make a bulk of renamings necessary. Likewise, a refactoring of the API, for instance so as to implement a given set of interfaces, may make changes of several method names and accessibilities necessary. Many of these scenarios will be difficult to capture in the form of a refactoring specification that can be reused, without significant modifications, across many different applications — yet, they describe rightful refactorings.

Compared to a traditional refactoring approach relying on a catalogue of standard refactorings (preferably ones for which implementations are available), ad hoc refactoring as we envision it has the following advantages:

- The user need not recognize that her/his intended refactoring corresponds to a standard refactoring, if such a standard refactoring does at all exists.
- In case no corresponding standard refactoring exists, the user need not decompose her/his intended refactoring into a series of standard refactorings, if this is at all possible.
- In case decomposition into standard refactorings is not possible, the intended refactoring need not be decomposed into a series of micro-refactorings, for which no standalone implementations may be available (so that they have to be performed by hand).

Ad hoc refactorings, or refactorings without names, are characterized by the fact that they are conceived and specified in an ad hoc fashion, with no reuse in mind. Their size is not a distinguishing criterion: it can range from very small (atomic or micro-refactoring) to big. Ad hoc refactorings are limited only by the capabilities of the underlying refactoring engine, which must be able to check if the specified refactoring is at all possible and, if necessary, to compute additional changes required to complete it.

3. CONCESSIONS TO REALITY

Our vision of ad hoc refactoring faces some non-trivial theoretical and practical challenges, so that to make it reality, we need to make some concessions.

- The Problem of Tractability Generally it is undecidable whether an arbitrary set of changes leaves a program’s behaviour unchanged. But even if we stay within the realm of decidability, analysing a set of changes for their effect on behaviour may be prohibitively expensive. Therefore, if we want to make ad hoc refactoring work, we have to restrict possible changes to ones whose effect we can control in all cases, at reasonable expense. Practically, this means that we will have to be conservative, i.e., will have to refuse refactorings even though they might be possible.

- The Problem of Eliciting the Refactoring Intent Another problem of our vision of ad hoc refactoring is to determine, from an arbitrarily long edit session between “begin refactoring” and “end refactoring”, the net changes that constitute the refactoring intent. This is particularly a problem since later edits of an edit session may change or undo earlier ones. At the same time, we have to filter from the edit session those changes that our refactoring engine cannot handle (cf. above), and reject them.

- The Problem of Determining the Additional Changes When the changes specified as the refactoring intent must be complemented by additional changes necessary to ensure behaviour preservation, the question arises whether these changes are reconcilable with the user’s intentions. This is indeed a tricky question, since differing from named refactorings such as PULL UP MEMBER, with which additional allowed changes can be directly associated (for instance, pulling up along other, referenced members), for ad hoc refactoring, the user’s intent beyond what (s)he stated explicitly can only be speculated about. On the other hand, the user will likely have no notion of which other changes might enable an intended refactoring that is otherwise impossible and, if asked upfront, will likely state that the changes specified with the refactoring intent are all that need be done. In case that this turns out to prevent the refactoring, (s)he will either adopt a trial-and-error approach of admitting additional changes, or will allow all conceivable changes right away, thereby giving up control over the refactoring (the performed additional changes).

All above problems have a common denominator: the kind and number of changes allowed. One way to attain control over changes is to represent all changes, user requested or computed by the refactoring engine, as setting selected properties of program elements to new values. For instance, the changes involved in the
5. EVALUATION

The experiment for selecting the additional changes the refactoring will perform starts with the specification phase where the user may freely enter properties to be changed, or select them by picking them out of a (similarly reusable) set of properties that the user may choose to be available.

The selected properties are then displayed to the user in a unique order. The user may either accept or change the order of the properties. The order is then used by the refactoring system to determine the sequence of changes. For instance, if the user selects the properties name, accessibility, and location, the refactoring system will first change the name of the element, then the accessibility, and finally the location. If the user changes the order of the properties to location, accessibility, and name, the refactoring system will first change the location, then the accessibility, and finally the name. This process is repeated until all properties are changed or the user cancels the refactoring.

The refactoring system then generates the changes that are required to achieve the selected properties. These changes are then presented to the user for review and approval. The user may accept or reject the changes, and the refactoring process is repeated until all changes are approved or the user cancels the refactoring.

4. IMPLEMENTATION

The implementation of the refactoring system is based on the model-driven architecture (MDA) and the Eclipse platform. The MDA provides a framework for the development of software systems, where the model is the core of the software development process. The Eclipse platform is a set of tools for the development of software, where the focus is on the development of tools for the MDA.

The refactoring system is implemented as a plugin for the Eclipse platform, which allows it to be integrated into the existing development environment. The plugin provides a graphical user interface (GUI) for the selection of properties to be changed, and a console interface for the execution of the refactoring.

The plugin also provides a set of rules for the generation of changes that are required to achieve the selected properties. These rules are implemented as Java classes and are executed by the plugin. The rules are based on the concepts of the refactoring framework and are designed to be reusable and extensible.

The plugin also provides a set of tools for the evaluation of the refactoring process. These tools allow the user to see the changes that are generated by the plugin, and to compare them with the changes that are generated by the refactoring engine.

The plugin is designed to be flexible and extensible, which allows it to be adapted to the needs of different development environments. The plugin is also designed to be scalable, which allows it to be used in large-scale development environments.
To answer this question, we have taken the constraint rules covering names, accessibilities, and locations (i.e., declaring types) of fields in Java (and thus the application scenarios of Section 2) and used them as the basis for our evaluation. To simulate the ad hoc refactoring that is at the heart of ad hoc refactoring, we assumed certain refactoring intents (modelled after the scenarios) and cast them into the following four evaluation scenarios:

1. Increase accessibility of 20 randomly picked fields. In case of ambiguous access resulting from this, allow fields to be renamed.
2. Swap names of two randomly picked fields.
3. Change the names of 20 randomly picked fields. In order to provoke conflicts with existing names, the new names are selected from the set of all variable identifiers. Conflicts may be resolved firstly by changing accessibility, and secondly by renaming other fields.
4. Change the names of 5 randomly picked fields to arbitrary new values, their location to a superclass, and their accessibility to public. Conflicts may be resolved as in evaluation scenario 3.

For comparability of results, all experiments were conducted with the same cascade of allowed changes, namely:
1. allow the change of references to fields in touched classes,
2. allow the change of references in touched packages,
3. allow the change of field declarations in touched classes, and
4. allow the change of references globally in that order.

Table 2 shows the results of applying each ad hoc refactoring 100 times on each program of Table 1 (except where stated otherwise, all numbers are averages). The results are interpreted as follows:

- Evaluation scenario 1 suggests that simple changes (such a increasing accessibilities) cause few problems in practice: The problem to be avoided here, that access of a field inherited from both an interface and a class becomes ambiguous [3], was never stumbled into. On the other hand, as can be seen by the number of generated constraints and times required, the precaution did not cost much, justifying the use of an ad hoc refactoring tool in even these simple cases.
- The picture is somewhat similar for a similarly innocuous refactoring, the name swap (evaluation scenario 2), the main difference being that it requires higher change levels (and thus more iterations) to be successful.
- When provoking more conflicts as in evaluation scenario 3, the cost both in terms of constraints generated and times needed increases substantially. At the same time, the success rates decrease (although by allowing the greatest number of additional changes, most refactorings can still be performed). When considering the longest time taken (more than 3 minutes), one needs to take into account that 20 fields have been renamed (giving us 10 seconds per field which, as a worst case, seems acceptable).

Table 1: Open source projects used in our evaluation

<table>
<thead>
<tr>
<th>PROJECT</th>
<th>FIELDS</th>
<th>PROJECT</th>
<th>FIELDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jaxen</td>
<td>315</td>
<td>Picocontainer V.1.3</td>
<td>128</td>
</tr>
<tr>
<td>Fit V 1.1</td>
<td>209</td>
<td>Commons Codec V.1.3</td>
<td>64</td>
</tr>
<tr>
<td>JUnit V. 3</td>
<td>161</td>
<td>Log4j V. 1.2</td>
<td>908</td>
</tr>
<tr>
<td>AntLR V. 3.2</td>
<td>238</td>
<td>HTMLParser V.1.6</td>
<td>507</td>
</tr>
<tr>
<td>Commons IO V. 1.4</td>
<td>170</td>
<td>JHotdraw V.6.0</td>
<td>693</td>
</tr>
</tbody>
</table>

Table 2: Results of simulated applications

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>EVALUATION SCENARIO</th>
<th>TOTAL (avg.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>required changes</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>success rate after</td>
<td>change level 1</td>
<td>100%</td>
</tr>
<tr>
<td>avg</td>
<td>17417</td>
<td>2375792</td>
</tr>
<tr>
<td>gen. const.</td>
<td>min</td>
<td>1212</td>
</tr>
<tr>
<td>avg</td>
<td>828</td>
<td>30</td>
</tr>
<tr>
<td>max</td>
<td>48054</td>
<td>17</td>
</tr>
<tr>
<td>max</td>
<td>506605</td>
<td>15</td>
</tr>
<tr>
<td>time required</td>
<td>min</td>
<td>0.033</td>
</tr>
<tr>
<td>avg</td>
<td>0.052</td>
<td>0.042</td>
</tr>
<tr>
<td>max</td>
<td>10</td>
<td>196</td>
</tr>
<tr>
<td>avg</td>
<td>100%</td>
<td>91%</td>
</tr>
<tr>
<td>avg</td>
<td>91%</td>
<td>79%</td>
</tr>
<tr>
<td>max</td>
<td>99%</td>
<td>93%</td>
</tr>
<tr>
<td>time required</td>
<td>avg</td>
<td>0.002</td>
</tr>
<tr>
<td>avg</td>
<td>14</td>
<td>146.2</td>
</tr>
<tr>
<td>min</td>
<td>2</td>
<td>1588.4</td>
</tr>
<tr>
<td>max</td>
<td>20837</td>
<td>248.6</td>
</tr>
</tbody>
</table>

Given the previous findings, it is comforting to see that the most complex ad hoc refactoring investigated, evaluation scenario 4, causes no extra cost — the main difference here is that the refactorings remain unsuccessful (i.e., are refused) more often than in all other scenarios. While this problem may be solved by adding more change levels, it poses the general question whether using change levels is always the best solution of elicting the allowed changes; an interactive mode, exploring possible solutions together with the user, may be more adequate here.

6. CONCLUSION

Starting from a set of practical refactoring scenarios, we have conceived a vision of ad hoc refactoring — or refactoring without names — and implemented it using constraint-based refactoring techniques. Results of a simulated application of our implementation of ad hoc refactoring suggest that our vision need not remain one.

7. ACKNOWLEDGMENTS

This work has been supported by DFG grant STE 906/4-1.

8. REFERENCES