Making Programmers Aware Of Refactorings

Peter Weißgerber Benjamin Biegel Stephan Diehl
University of Trier, 54286 Trier, Germany
{weissger, diehl}@uni-trier.de benjamin.biegel@gmail.com

Abstract

Modern integrated development environments, such as ECLIPSE, provide automated or semi-automated refactoring support. Despite this support, refactorings are often done manually — either because the developer is not aware of the tool support or because he is not aware that what he did was a refactoring. Moreover, as we found in [7] programmers tend to do a bad job in documenting the refactorings they have performed or even do not document them at all.

In this paper we present the design rationale of a plug-in for the ECLIPSE development platform that informs programmers about the refactorings they have performed manually and provides hyper-links to web sites describing these. The plug-in is currently under development. Finally, it should support the developer in documenting refactorings by appending an exact description of each performed refactoring to the CVS/SVN log message. For such refactorings that have been done manually, but can be performed automatically using ECLIPSE, our tool should inform the developer that this tool support exists and it is much safer to use it than to implement the refactoring manually.

1. Introduction

In evolving software systems, refactoring tasks are virtually essential too keep the code maintainable and the code structure understandable, and thus, are part of the daily work of a developer [6]. However, if done manually, refactorings can be error-prone. Thus, Fowler's book [3] contains a large catalog of refactorings and for each of it a description how it is implemented correctly. An additional problem is that, as we found in earlier work [7], for a project such as TOMCAT3 less than 10% of all refactorings are documented in the log messages of the software repository.

Thus, a tool that makes the programmer aware of his refactorings, provides links to web pages explaining these, and helps to document these, would certainly be helpful.

2. Refactoring Identification

This section gives a short overview on our refactoring detection approach, which is described in detail in [7]. In summary, this approach works in three phases:

Preprocessing: Find out which code blocks (classes, fields, methods; all identified by their fully-qualified signature) have been added and deleted compared with an earlier version of the software (e.g., the latest version in the repository).

Signature-based Identification: Compare the added and removed code-blocks using a signature-based approach to find refactoring candidates. E.g., if the method `doComputation(int, int):double` has been removed and the method `compute(int, int):double` added to the same class, we detect a candidate for a Rename Method refactoring.

Ranking and Filtering: Rank the candidates using code clone detection on the old resp. new body of the block. Filter out all candidates below a certain rank.

The remaining candidates are presented to the user. Obviously, the quality of these candidates strongly depends on the exact configuration of the clone detection and the filter. However, discussing these configuration details is beyond the scope of this paper. Instead, we focus on how to leverage the information about identified refactorings to make programmers aware of refactorings, improve the documentation of refactorings in log messages, and help to prevent errors.

Currently, we detect move, as well as rename refactorings, changes to the visibility of a symbol, and parameter additions/deletions to/from methods. Additionally, we want to record all refactorings that are done using the ECLIPSE refactoring functionality.

3. Integration in ECLIPSE

In the following we describe how we intend to integrate our refactoring identification approach into ECLIPSE.

A Refactoring View in ECLIPSE

As the space in this paper is very limited, we do not describe each feature separately. Instead, we illustrate how our tool could look like and how it can be used by a programmer by means of an example.

Figure 1 shows a mock-up how the user interface of our ECLIPSE plug-in could look like. The list of identified refactorings is presented in its own view next to the list of problems, declarations, the console, etc.. The first line contains a summary of how many refactorings have been identified, and how many of those have been performed manually respectively automatically (using the ECLIPSE refactoring tool).

In the table below the summary, each line shows information about a single identified refactoring. The first column of each line contains the refactoring kind, while the second column contains a detailed description of the refactoring. The third column indicates whether this refactoring has been done automatically or manually. Next, links to web pages with additional information on this kind of refactoring are listed. For example, we can link to the particular sub-page below `www.refactoring.com` (the web page of the book [3] which also includes the refactoring catalog). The last

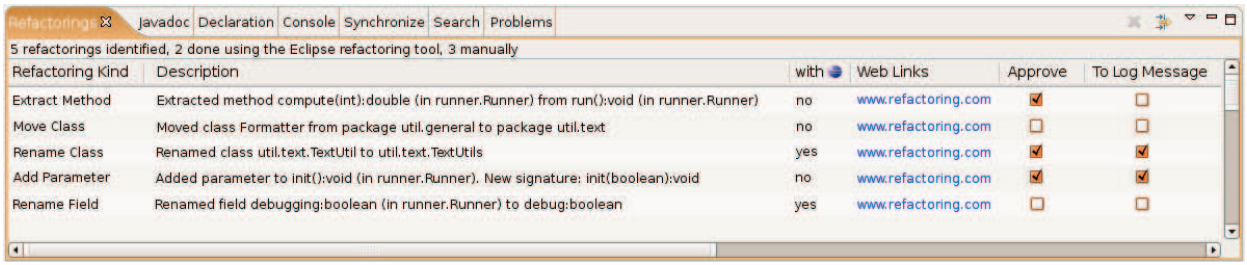

Figure 1. Example how the Eclipse integration could look like

two columns provide check-boxes to approve the refactoring and to automatically add a description of it to the log message when it is committed to the repository.

To make programmers really aware of new refactorings we also intent to show a pop-up every time new refactorings have been identified. Obviously, such a pop-up can annoy developers who are experienced enough to watch the refactoring list anyway or who are not interested in this functionality. Thus, this pop-up should be enabled by default but may be disabled easily.

Options

In the next paragraphs, we discuss technical issues respectively design options.

When to update and show the refactoring list? There are several options when the refactoring list can be updated: Our first idea was to update the list *on request*, i.e. every time the user pushes a particular button. The advantage would be that the computation of the refactoring candidates (which may cause latency on slow systems) is only done when the user requests it. However, requiring the programmer to push a button conflicts with our goal to make him aware of refactorings. Thus, we dismissed this approach and decided to update the list automatically *on save*, that means each time a file is saved to the disk. If the pop-up option is enabled, also a pop-up window opens then, provided a new refactoring is detected. Finally, the third option is that the list of refactorings is updated and presented when the developer tries to commit his changes to the repository. If only this option (and not "on save") is used, the problem is that the developer is informed about his changes quite late, but at least before they get available to other developers. We decided to implement this option additionally to "on save": if the pop-up is enabled, it should be shown again before the commit operation is executed.

How to get the changes? As explained in Section 2 we need to compare removed code block with added code blocks in order to identify refactorings. Thus, obviously the current version of the software (as shown in the workspace) has to be compared with some older version. The first approach would be to take the older version from the local history of the changes within ECLIPSE [1]. This approach should work quite fast and be easy to implement because no access to the software repository (which could be a SCM system like CVS or SUBVERSION) is required. However, as the memory to store this local change history is limited within ECLIPSE, old changes in a session can get lost. Furthermore, the local history is cleared whenever ECLIPSE is restarted. Thus, retrieving the older version from the repository enables us to identify more refactorings under certain conditions.

[1] The local history is an ECLIPSE feature that allows to browse through the n latest changes performed in an ECLIPSE session.

4. Related Work

While there exists tool support for performing refactorings in most of today's programming IDEs [5] only few researchers have tried to identify refactorings automatically [2, 7, 1].

Henkel and Diwan have shown that is useful to record automatically performed refactorings [4]. Their CATCHUP tool records refactorings performed with the ECLIPSE tool and allows to reapply these to client code. In contrast, we additionally take identified refactorings into account and help the programmmer keeping track of the refactorings he has done.

5. Conclusion

In the introduction we have motivated why an ECLIPSE plug-in that makes programmers aware of refactorings and helps to document these, would be a helpful tool. We explained how the refactoring identification works and presented how it can be integrated reasonably into ECLIPSE.

While the refactoring detection algorithm has already been implemented and evaluated [7], we have just recently started to implement the ECLIPSE integration. We are very confident, that we will be able to present a first prototype at the workshop provided that the paper is accepted. This prototype should at least perform the refactoring identification and present a list of the identified refactorings including web links, each time a developer saves his changes.

References

[1] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proc. Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000)*.

[2] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Finding Refactorings via Change Metrics. In *Proc. European Conference on Object-Oriented Programming (ECOOP 2006)*.

[3] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2001.

[4] J. Henkel and A. Diwan. Catchup!: capturing and replaying refactorings to support api evolution. In *Proc. International Conference on Software Engineering (ICSE 2005)*.

[5] E. Mealy and P. Strooper. Evaluating software refactoring tool support. In *Proc. Australian Software Engineering Conference (ASWEC 2006)*.

[6] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions On Software Engineering*, 30(2), 2004.

[7] P. Weißgerber and S. Diehl. Identifying Refactorings from Source-Code Changes. In *Proc. International Conference on Automated Software Engineering (ASE 2006)*.