Automated Refactoring Suggestions Using the Results of Code Analysis Tools

Steffen Herbold, Jens Grabowski
Institute of Computer Science
University of Göttingen
Göttingen, Germany
Email: {herbold, grabowski}@cs.uni-goettingen.de

Helmut Neukirchen
Faculty of Industrial Engineering,
Mechanical Engineering and Computer Science
University of Iceland
Reykjavík, Iceland
Email: helmut@hi.is

***Abstract*—Static analysis tools are used for the detection of errors and other problems on source code level. The detected problems related to the internal structure of a software can be removed by source code transformations called refactorings. To automate such source code transformations, refactoring tools are available. In modern integrated development environments, there is a gap between the static analysis tools and the refactoring tools. This paper presents an automated approach for the improvement of the internal quality of software by using the results of code analysis tools to call a refactoring tool to remove detected problems. The approach is generic, thus allowing the combination of arbitrary tools. As a proof of concept, this approach is implemented as a plug-in for the integrated development environment Eclipse.**

***Keywords*—Software verification and validation; Software inspection techniques; Software testing tools; Refactoring; Tool integration**

I. Introduction

An important constituent of software verification and validation is the inspection of source code by means of static analysis tools. These tools are able to detect many possible defects, as well as coding problems on the source code level. In particular, issues that relate to bad internal structure are called *code smells* [1] or just *smells*. Code smells can be removed by applying *refactorings*. Refactoring is defined as "*a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior* [1]". Modern *integrated development environments* (IDE), such as the *Eclipse Java Development Tools* (JDT) [2], are able to apply the transformation steps of a refactoring automatically. Thus, the risk of unintentionally changing the behaviour of a program during the application of a refactoring is significantly reduced.

While refactoring is the natural means to remove code smells revealed by static analysis and even though both static analysis and refactoring are well supported and automated by tools, there is a gap between these tools, i.e., suggesting the application of automated refactorings based on the results of static analysis tools. To overcome this gap, we present a generic approach to connect static analysis tools and refactoring tools. As proof of concept, we have implemented our approach on top of the *Eclipse* platform [3] as a plug-in called *AddFix*. AddFix allows generic and flexible association of refactorings from various Eclipse-based refactoring tools to results from arbitrary static analysis tools that are available as plug-ins for Eclipse.

The structure of this paper is as follows. After this introduction, related work is discussed in Section II. In Section III, we present our general approach. Subsequent, in Section IV, we describe the application of our approach and its implementation as AddFix plug-in for the Eclipse platform. An evaluation of the applicability of our approach is provided in Section V. Finally, a conclusion and an outlook on future work are given.

II. Related Work

A multitude of tools support the automated transformation of source code using refactorings, e.g., the *Refactoring Browser* for Smalltalk source code [4] or the Eclipse JDT and *NetBeans IDE* [5] for Java source code [2]. A survey on refactoring and refactoring tools is provided by Mens and Tourwé [6].

Other tools are able to detect code smells using static analysis, e.g., *FindBugs* [7][8] or *PMD* [9][10] for Java source code. However, these tools (and their analysis results) are not linked to refactoring tools.

There are several theoretical approaches dealing with how entities that need refactoring can be automatically detected. Crespo et al. [11] have suggested a language independent metric-based approach to detect code smells in object-oriented software, which can then be used to infer where refactorings should be applied. Also, Simon et al. [12] have shown that software metrics can be used to detect entities to be refactored. Kataoka et al. [13] suggest an invariant-based approach to detect where refactorings can be applied. Balazinska et al. [14] suggest clone-analysis to support detection of duplicated code that needs to be refactored. However, these approaches only propose refactorings, without integration in a refactoring tool to support their automated application.

Mens et al. [15] have shown that it is possible to use code smell detection as an automated method for the suggestion of refactorings and integrated their approach with the Smalltalk Refactoring Browser. Neukirchen et al. [16] have developed the quality assurance tool TRex for the test language TTCN-3. TRex uses software metrics and code smell detection to automatically suggest and apply one of the implemented refactorings. The Eclipse JDT [2] supports the application of

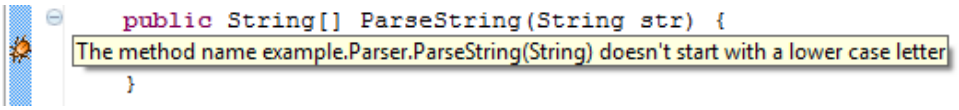

Fig. 1. This figure shows a marker that was generated by the FindBugs Eclipse plug-in: The tool FindBugs used static analysis to detect the problematic method name and placed a marker near the corresponding source code line to indicate the problem.

fixes mainly to remove compilation problems of Java source code. (Strictly speaking, these problems are not smells and the fixes are not behaviour preserving refactorings.) All these approaches have in common that they are hard-coded for the respective refactoring tool and hard-linked to the respective analysis components. In contrast, the approach presented in this paper is language independent and generic, thus allowing to couple arbitrary analysis tools with arbitrary refactoring tools.

III. Approach

Our approach to combine the results of static analysis tools with refactoring tools is based on the fact that the result generated by code analysis tools typically includes textual descriptions of the problems that were detected. Thus, we use regular expressions to match these textual descriptions. The Eclipse and NetBeans platforms, e.g., display the results of static code analysis tools in form of *markers* (see Figure 1) or *error stripes* respectively. Both contain a text message, which can be matched using regular expressions.

In our approach, rules are used to associate refactorings to the textual problem descriptions. In addition to a regular expression, a rule consists of a reference to a refactoring implementation and a specification of the parameters that need to be passed to this refactoring. To keep the reference to the refactoring implementation independent from a specific refactoring tool, an adapter [17] rather than an actual refactoring implementation is referenced from within the rule. By exchanging the adapter, different refactoring tools can be targeted.

IV. The AddFix Tool

We have developed *AddFix* [18] as proof of concept implementation for our approach. While it is based on the notion of *markers* and *quick fixes* of the Eclipse platform, it is completely independent from the analysis tools and the refactoring tools that shall be coupled. The only requirement is that the analysis tool reports its results by generating Eclipse markers and that the refactoring tool can be called from within an Eclipse plug-in, i.e., our AddFix tool that implements a quick fix. Both the analysis tools and the refactoring tools are available as plug-ins for the Eclipse platform.

In this section, the relevant Eclipse concepts are introduced. Afterwards, we describe which problems needed to be solved by AddFix and how AddFix solves them.

A. Eclipse Concepts

In Eclipse, a quick fix is an automated solution to a problem. If, e.g., within a Java source code, an entity is referenced that is externally defined but not imported, a quick fix could automatically add the missing import statement. Using an *extension point*, it is possible to define and add own quick fixes. Extension points define where and how plug-ins can add functionality to Eclipse. There are two types of extension points: those to which contributions can only be made at compile-time and those to which contributions can also be made at run-time. Unfortunately, the extension point to define quick fixes belongs to the first category. This restricts the capabilities to add quick fixes by an end user at run-time.

In Eclipse, markers are a versatile concept. Generally, they can be used to mark resources such as files or projects almost arbitrarily. Every marker defines at least three attributes: a unique *Id*, a *type* and a *message*. The unique Id is used as an internal identifier by Eclipse. The type and the message define how the marker should be interpreted. For example, the type of a marker could be “problem” and its message “`Semicolon missing`”. The type of a marker is also important because quick fixes cannot be added to any marker type, but only to those, that need “fixing”. An example for markers that do not need to be fixed are markers of type “task”. By definition, a task cannot be fixed, but has to be performed. Thus, Eclipse does not allow quick fix support for task markers.

Most markers have more attributes than only the above mentioned three. Typically, specific information about the location of a problem is also provided by a marker. The above example of a marker for a missing semicolon would not be useful, if it were only associated with a file name but the location of the missing semicolon would still be unclear. However, if the line would be known, the problem would be easy to locate. The marker shown in Figure 1 also contains information about the file and the line number as well. Hence, Eclipse is able to display this marker next to the corresponding line of the affected file.

In general, the ability to locate and remove the actual reason for a marker depends on the additional information provided by the marker. In sections IV-C and IV-D, we will show how marker information can be exploited in a general way to allow the addition of quick fixes to markers by utilising the information that can be obtained from a marker.

B. Associating Quick Fixes to Markers

To implement our approach for linking Eclipse-based static analysis tools and refactoring tools together, we exploit the Eclipse quick fix mechanism. Quick fixes that call refactoring implementations are used to combine the two kinds of tools. To achieve this, two general problems need to be solved:

First, it has to be decided which quick fix, i.e., which kind of refactoring, to add to which marker. To support arbitrary static

```
1  <addfix>
2    <rule
3      markerText="The .* name .* doesn't start with a lower case letter">
4      <fix>
5        <class>
6          de.ugoe.cs.swe.addfix.quickfix.java.RenameLowerCase
7        </class>
8        <param desc="NAME">
9          <pattern>
10           The .* name (.*) doesn't start with a lower case letter
11         </pattern>
12       </param>
13       <param desc="TYPE">
14         <pattern>
15           The (.*) name .* doesn't start with a lower case letter
16         </pattern>
17       </param>
18     </fix>
19   </rule>
20 </addfix>
```

Fig. 2. The definition of a rule in XML: The regular expression that defines where the rule is applicable is specified in the `markerText` attribute of the `rule` node. The `fix` subnode(s) specifies the quick fixes that will be added to markers that match the rule: the node `class` defines the class in which the quick fix is implemented. The optional `param` node(s) define the parameters that are passed to the quick fix.

analysis tools and refactoring tools, this association should not be hard-coded, but user configurable at run-time. Hence, a strategy is needed that is flexible enough to define a set of rules at run time – simple enough to be feasible, but also specific enough such that the quick fixes will only be added where they are applicable and useful. This is described in Section IV-C.

Second, to call a refactoring implementation when a quick fix is invoked, all the data required for a refactoring needs to provided by the quick fix instance to the refactoring implementation. When creating an instance of a quick fix, this data needs to be extracted from the information provided by a marker. Consider, e.g., a *Rename* refactoring: Such a refactoring requires information about the identifier of the entity that shall be renamed. Usually, this information can be easily obtained if the marker provides sufficient information, like the location (line, column) of the identifier within a file. On the other hand, consider an *Extract Method* refactoring for the extraction of a part of a very long method into a method on its own. While it is easy to identify long methods, it is generally impossible to automatically decide what exact part of a method shall be extracted. Whether the information provided by a marker is sufficient to initiate a refactoring depends on the particular marker information and the refactoring to be associated. A generic and flexible approach to extract this information from a marker is presented in Section IV-D.

C. Definition of a Rule Set

Regular expressions are used to allow users to flexibly specify which quick fixes are added to a marker generated by an analysis tool: if the text of a marker matches the regular expression, the quick fix is applicable. For example, a rule containing the regular expression "`The .* name .* doesn't start with a lower case letter`" (Line 3 in Figure 2) would match markers with the message "`The method name ExampleMethod doesn't start with a lower case letter`" as well as markers with the message "`The field name ExampleField doesn't start with a lower case letter`" and could suggest a Rename refactoring as an associated quick fix. On rule level, the quick fix to use is specified by referring to the name of a Java class (Line 6 in Figure 2). This class serves as an adapter to call the actual refactoring implementation provided by a specific refactoring tool. This approach allows easy and flexible definition of rules. For example, new quick fixes that call refactorings can be added by simply adding the class defining the quick fix. The users of the AddFix tool need not to care about the Eclipse extension point that is internally responsible to determine whether the quick fix is applicable or not, but simply solves this by using a regular expression. In the same manner, existing quick fixes can simply be added to markers to which they are not yet associated by adding a corresponding AddFix rule.

As mentioned in Section IV-A, Eclipse does not allow to add quick fixes to markers of type "task". This is a drawback, since a task can sometimes be performed by a quick fix. An example for this is the static analysis tool PMD: for the problems it detects, PMD creates task markers instead of markers of the type "warning". To resolve this, AddFix supports not only to add automatically quick fixes to markers, but also to add automatically new warning markers to existing task markers. Again, rules based on regular expressions are used. These expressions are matched against the texts of all task markers. If there is a match, AddFix adds a new warning marker with the same attributes as the task marker. Then, quick fixes can be automatically added to the newly created warning markers based on the AddFix rules.

Due to the inflexibility of the Eclipse extension point which is used to add quick fixes to markers, AddFix can only add quick fixes to markers, whose types were known at compile time (see Section IV-A). A workaround for this restriction

of the Eclipse platform would be to let AddFix create new markers for all unknown marker types in the same way it is already done for task markers.

D. Using Quick Fixes to Call Refactorings

To apply a refactoring, a sufficient amount of information concerning the entity that shall be refactored is needed, e.g., the location in the source code where to apply a refactoring. Since our approach relies solely on the markers generated by analysis tools, all the information required for the application of a refactoring needs to be obtained from these markers.

As described in Section IV-A, only very few marker attributes are mandatory. While markers generated by analysis tools typically contain information concerning the file and the line number to which a marker refers, often further information concerning the start and end column of a problem is lacking. However, for applying most refactorings, more detailed information is required. Consider, e.g., a marker with the text "`The method name example.Parser.ParseString(String) doesn't start with a lower case letter`" as it is shown in Figure 1. From the message text, it can be inferred that a Rename refactoring is required to remove the problem. However, for the application of a Rename refactoring, a refactoring tool typically needs a pointer to the identifier to be renamed, e.g., by specifying the line and column location of that identifier within a file. Unfortunately, the marker shown in Figure 1 contains only line and file name information, but no column information. As multiple identifiers may be contained in a single line, knowing just the line number of a marker is in general not sufficient to select the correct identifier to be renamed.

However, from the marker text, further information can be extracted: The marker text provides information that a method name (and not, e.g., a field name) has to changed and it contains detailed information about the package, class, name and parameter types of the method to be renamed. This information is sufficient to identify any method within a project unambiguously. (In fact, not even the file name and the line number information would be required in this case.)

To be able to pass this information to a quick fix instance that calls a refactoring, our rule approach provides a flexible parameter mechanism that allows the user of AddFix to specify which parts of a marker text to extract and to pass as parameters to a quick fix. As a result, AddFix is applicable even if the optional marker attributes concerning the location are either missing or wrong (provided that the marker text itself provides sufficient information).

AddFix rules that extract information from marker texts and pass this information as named parameters to quick fixes, use regular expressions also to specify which parts of a marker text to extract: those parts of a regular expression that are enclosed by parentheses determine which parts of the marker text are extracted and passed as an actual parameter value. In the example rule in Figure 2, the regular expression "`The .* name (.*) doesn't start with a lower case letter`" in line 10 would yield the actual parameter value "`example.Parser.ParseString(String)`" when applied to the marker shown in Figure 1. To support passing multiple parameters to a quick fix, parameters are named: In Figure 2, lines 8–12 define a parameter named "`NAME`", whereas lines 13–17 define a parameter named "`TYPE`". The latter uses a regular expression to extract those parts of the marker text that specify whether a method name or a field name is affected (Line 15).

In addition to parameter values extracted at run-time from marker texts, AddFix supports also the specification of rules that provide constant parameter values. This allows to re-use and configure an existing quick fix class from within different rules just by passing appropriate parameters.

The quick fix classes that are called by an AddFix rule will in turn evaluate the parameters that are passed to call the corresponding refactoring implementation of the specific refactoring tool. One might argue, that the task of extracting information from marker texts can also be performed directly by the quick fix classes. However, this would make quick fix classes directly dependent on specific marker texts or analysis tools that generate these specific marker texts. In this case, to support additional analysis tools, the quick fix would need to be changed. Using our parameter mechanism, it is possible to define the parameters together with the marker-specific rules, independent from the generic quick fix classes that are only specific with respect to the refactoring tool to call, but not with respect to the analysis tool generating the markers.

E. Implementation

The AddFix Eclipse plug-in we implemented as proof of concept is divided into three parts: the core; the user interface; the quick fixes.

The core is responsible for the rule management and rule application. This includes the responsibility to add new markers when matching task markers, as well as the removal of all markers set by AddFix. The rules are stored persistently using an XML file. Figure 2 shows how rules are defined in the XML file. The integrity of the rule set is guaranteed by a *Document Type Definition* (DTD) grammar.

The user interface package provides a preference page to manipulate the rule set, as it is shown in Figure 3. It allows easy adding, editing and deleting of rules.

The quick fix package provides quick fix implementations that serve as adapters to call refactorings offered by different Eclipse refactoring plug-ins. In addition, an abstract class is provided for adding quick fixes that can use the parameter mechanism of AddFix. The abstract class defines the required functionality needed to use the parameter mechanism. Section V-A discusses a quick fix implementation that uses the parameter mechanism.

V. Evaluation of AddFix

To evaluate the applicability of AddFix, we conducted two case studies where we used AddFix to couple different static analysis tools available as Eclipse plug-in with different

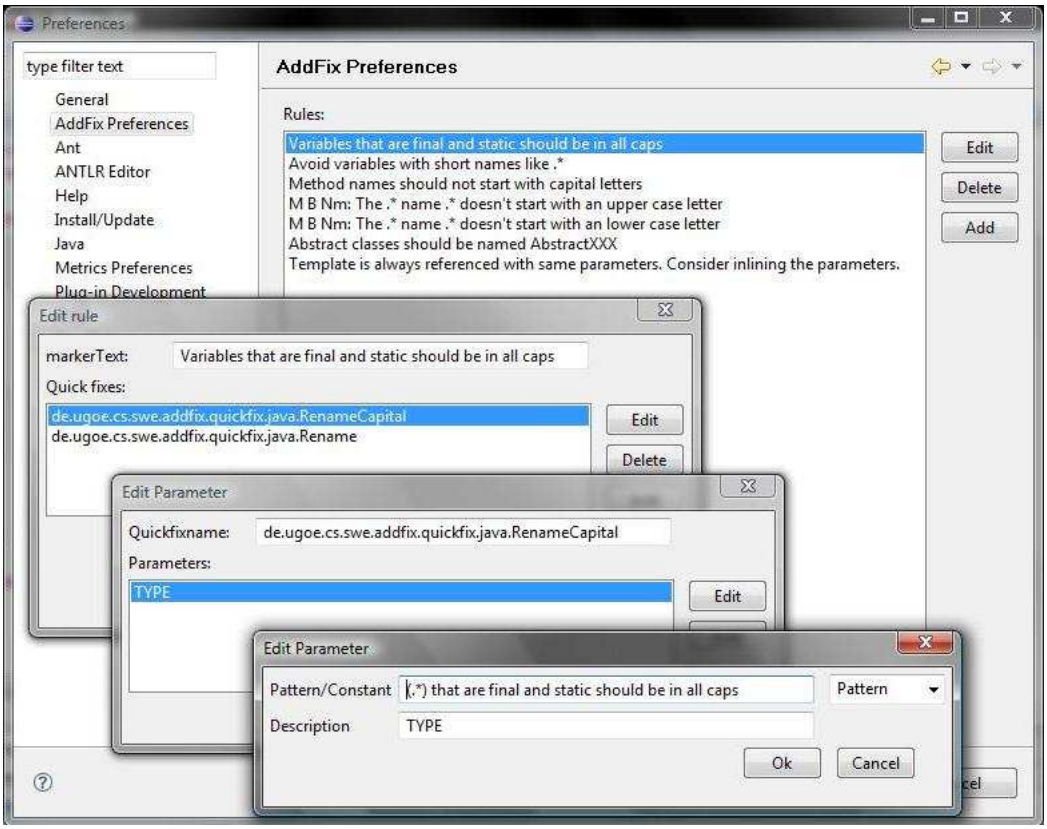

Fig. 3. This figure shows how the AddFix rules can be edited using an Eclipse preference page: The rules are listed showing their regular expression. Using additional edit dialogs, the class that defines the associated quick fix and the parameters can be edited.

Eclipse refactoring plug-ins. As a part of this case study, refactoring-tool-specific quick fixes were implemented and analysis-tool-specific AddFix rules were written.

The results of these case studies demonstrate that AddFix is both effective and efficient: associating quick fixes to markers using the regular expression-based AddFix rules was feasible. By adding quick fixes to markers, either a selective application of individual quick fix instances is offered by the Eclipse platform or all quick fixes of the same class can be applied to a set of problems in a batch (see Figure 4). In both cases, the corresponding refactoring implementation is called that is correctly configured by each quick fix instance. As a result, the code smells reported by the analysis tool are efficiently removed. In contrast, had no quick fixes been associated to the markers, a user would have had to decide on the appropriate refactoring and call it individually for each marker. Thus, the AddFix approach significantly reduces the efforts for the removal of problems detected by analysis tools.

In the first case study, AddFix is applied to connect Java-specific analysis and refactoring Eclipse plug-ins. In the second case study, AddFix is used to connect markers and refactorings specific to an Eclipse-based tool for the test language TTCN-3.

A. Java Case Study

We used Eclipse plug-ins of the FindBugs tool [8] and the PMD tool [10] to analyse Java source code and generate corresponding markers. Both tools provide only the line number, but no column information in their generated markers. Furthermore, PMD creates only task markers for which no addition of quick fixes is supported by Eclipse. We have written AddFix rules for several marker texts that refer to code smells that can be resolved using a Rename refactoring. The AddFix parameter mechanism is used to pass additional information to the quick fix classes. An example rule for FindBugs is shown in Figure 2. To support the addition of quick fixes to PMD task markers, additional rules for the creation of corresponding warning markers have been written.

The Eclipse JDT provides a flexible Rename refactoring implementation. Several quick fix classes have been implemented to rename identifiers using different capitalisation as required by the coding rules assumed by the analysis tools. The JDT Rename refactoring can be applied to any Java element by passing its identifier as stored in the *Abstract Syntax Tree* (AST) internally used by the JDT. Thus, if a quick fix wants to call this refactoring, it has to be able to locate the Java element in the AST. To be able to work with as many different markers and analysis tools as possible, the quick fix class implemented for this case study uses both optional marker attributes and the parameter mechanism.

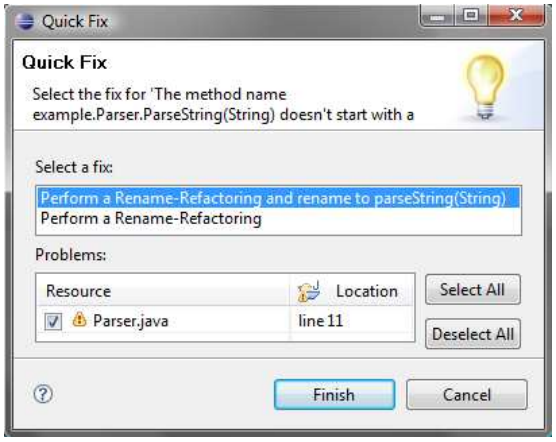

Fig. 4. An Eclipse dialog to apply a quick fix to a set of applicable markers: The quick fixes were added by AddFix to the marker shown in Figure 1.

If the exact position of the element in the source code is provided by an optional marker attribute, the quick fix class can easily obtain the identifier using the capabilities of the JDT. In addition, the quick fix class can also use the combination of the line number and either the type or the name of the Java element to locate the identifier to be renamed. A further parameter the quick fix will use if provided, is a "look-around" parameter to deal with the fact that some analysis tools tend to place their markers one line off (see Figure 1 where the marker has been wrongly created one line below the actual method name). To deal with this, the quick fix class can use the look-around parameter to consider also n lines before and after line number provided by a marker. The value n of the look-around parameter is defined as an analysis tool-specific fixed parameter value in the AddFix rules.

B. TTCN-3 Case Study

To show that AddFix is also able to target other languages and tools than those from the Java domain, we applied it also to the test language TTCN-3 that is supported by the Eclipse plug-in TRex [19]. TRex comes with an analysis component that creates markers and provides a refactoring component as well. While TRex already associates refactorings via quick fixes for some of its markers, we used AddFix to add further associations. In addition to corresponding AddFix rules, a quick fix class has been created that calls the *Inline Template Parameter* refactoring of TRex. This quick fix is implemented specific to the TRex markers, because these markers already contain required pointers into the TRex AST. Hence, the implementation of this tool-specific quick fix was very easy.

VI. Conclusion and Future Work

We have presented a flexible and tool-independent approach to close the gap between code analysis tools and refactoring tools by associating them to each other. This approach has been validated by our open-source implementation called AddFix [18]. With our approach, it is possible to not only suggest, but also trigger and automatically call refactorings based solely on information provided by code analysis tools. Hence, our approach and its implementation can be used as an adapter between the already existing code analysis tools and refactoring tools. To reduce the dependency on tool-specific marker attributes, a method to extract information from marker texts has been developed. Furthermore, AddFix can be used to write quick fixes for Eclipse without having to deal with the Eclipse extension point for quick fixes. Thus, AddFix reduces the complexity of adding own quick fixes to Eclipse.

While the AddFix implementation is specific to the Eclipse platform, the underlying approach is applicable to other platforms that support concepts similar to Eclipse markers and quick fixes.

In our future work, we will evaluate the applicability of AddFix to other code analysis tools, e.g., CheckStyle [20], and other tools for fixing problems will be evaluated. Furthermore, we want to analyse which problems can be fixed by these tools automatically based on the information static analysis tools provide and implement corresponding flexible and re-usable quick fixes for these problems.

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[2] Eclipse Foundation, "Java Developement Tools Eclipse Plug-in," 17.06.2009. [Online]. Available: http://www.eclipse.org/jdt/

[3] ——, "Eclipse Project," 17.06.2009. [Online]. Available: http://www.eclipse.org

[4] D. Roberts, J. Brant, and R. Johnson, "A refactoring tool for Smalltalk," *Theory and Practice of Object systems*, vol. 3, no. 4, pp. 253–263, 1997.

[5] Sun Microsystems, "NetBeans," 17.06.2009. [Online]. Available: http://www.netbeans.org

[6] T. Mens and T. Tourwe, "A survey of software refactoring," *IEEE Trans. Softw. Eng.*, vol. 30, no. 2, pp. 126–139, 2004.

[7] B. Cole, D. Hakim, D. Hovemeyer, R. Lazarus, W. Pugh, and K. Stephens, "Improving your software using static analysis to find bugs," in *Dynamic Languages Symposium 2006*. ACM, 2006.

[8] "FindBugs," 17.06.2009. [Online]. Available: http://findbugs.sf.net

[9] T. Copeland, *PMD applied*. Centennial Books, 2005.

[10] "PMD," 17.06.2009. [Online]. Available: http://pmd.sf.net

[11] Y. Crespo, C. Lopez, R. Marticorena, and E. Manso, "Language independent metrics support towards refactoring inference," in *9th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 2005.

[12] F. Simon, F. Steinbruckner, and C. Lewerentz, "Metrics based refactoring," in *Fifth European Conference on Software Maintenance and Reengineering*. IEEE, 2001.

[13] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin, "Automated support for program refactoring using invariants," in *IEEE International Conference on Software Maintenance 2001*. IEEE, 2001.

[14] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Advanced clone-analysis to support object-oriented systemrefactoring," in *7th Working Conference on Reverse Engineering*. IEEE, 2000.

[15] T. Mens, T. Tourwe, and F. Munoz, "Beyond the refactoring browser: Advanced tool support for software refactoring," in *Sixth International Workshop on Principles of Software Evolution*. IEEE, 2003.

[16] H. Neukirchen, B. Zeiss, J. Grabowski, P. Baker, and D. Evans, "Quality assurance for TTCN-3 test specifications," *Software Testing, Verification and Reliability (STVR)*, vol. 18, no. 2, pp. 71–97, Jun. 2008.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[18] "AddFix," 17.06.2009. [Online]. Available: http://gforge.informatik.uni-goettingen.de/projects/addfix/

[19] "TRex," 17.06.2009. [Online]. Available: http://www.trex.informatik.uni-goettingen.de/

[20] "CheckStyle," 17.06.2009. [Online]. Available: http://checkstyle.sf.net