FaultBuster: An Automatic Code Smell Refactoring Toolset

Gábor Szőke, Csaba Nagy, Lajos Jenő Fülöp, Rudolf Ferenc, and Tibor Gyimóthy
Department of Software Engineering
University of Szeged, Hungary
{gabor.szoke,ncsaba,flajos,ferenc,gyimothy}@inf.u-szeged.hu

***Abstract*—One solution to prevent the quality erosion of a software product is to maintain its quality by continuous refactoring. However, refactoring is not always easy. Developers need to identify the piece of code that should be improved and decide how to rewrite it. Furthermore, refactoring can also be risky; that is, the modified code needs to be re-tested, so developers can see if they broke something. Many IDEs offer a range of refactorings to support so-called automatic refactoring, but tools which are really able to automatically refactor code smells are still under research.**

In this paper we introduce FaultBuster, a refactoring toolset which is able to support automatic refactoring: identifying the problematic code parts via static code analysis, running automatic algorithms to fix selected code smells, and executing integrated testing tools. In the heart of the toolset lies a refactoring framework to control the analysis and the execution of automatic algorithms. FaultBuster provides IDE plugins to interact with developers via popular IDEs (Eclipse, Netbeans and IntelliJ IDEA). All the tools were developed and tested in a 2-year project with 6 software development companies where thousands of code smells were identified and fixed in 5 systems having altogether over 5 million lines of code.

***Index Terms*—Automatic refactoring, code smells, coding issues, antipatterns, SourceMeter, Columbus**

I. Introduction

As the process of *refactoring* [1], [2] has become more and more recognized by developers and researchers, more tools have become available to support it. Also, many IDEs offer refactoring features to provide assistance in these regular tasks. Eclipse, for instance, has a separate *Refactor menu* where such operations are available, e.g., developers can rename a source code element (e.g., a variable) and the IDE will correct all its references. Similarly, it is possible to extract local variables, methods, classes, or move elements, among several other operations. These, so-called *automatic refactorings* help developers to easily rewrite parts of the source code of a software system in order to improve its quality without modifying the observed external functionality.

Tools which support automatic refactorings often assume that programmers already know how to refactor and they have knowledge about the catalog of refactorings [2], but this is usually not a reasonable assumption. As Pinto et al. found it in their study where they examine questions of refactoring tools on Stack Overflow, programmers are usually not able to identify refactoring opportunities, because of lack of knowledge in refactoring, or lack of understanding of the legacy code. They also claim that "*refactoring recommendations is one of the features that most of Stack Overflow users desire (13% of them)*" [3]. In another recent study, Fontana et al. compare the capabilities of refactoring tools to remove code smells and they identify only one tool (*JDeodorant*) which is able to support code smell detection and then to suggest which refactoring to apply to remove the detected smells [4]. Certainly, most current tools lack this required feature to identify refactoring opportunities and to recommend problem-specific corrections which could be even fully automatically performed by the tools (or semi-automatically including some interactions with the developers).

In this paper, we introduce *FaultBuster*, an automatic code smell refactoring toolset which was designed with the following primary goals:

- to assist developers in identifying code smells that should be refactored,
- to provide problem specific, automatic refactoring algorithms to correct the identified code smells (currently it implements 40 refactoring algorithms),
- to integrate easily with the development processes via plugins of popular IDEs (Eclipse, NetBeans, IntelliJ) so developers can initiate, review, and apply refactorings right in their favorite environments.

The development of FaultBuster was carried out in an EU-supported project in which 6 software development companies were involved. Most of these companies were founded before the millennium and developed Java systems for different ICT areas like ERPs or Collection Management Systems. Their Java projects consisted of over 5 millions lines of code altogether. So they were not just involved in the development of FaultBuster, but all the companies really wanted to improve the quality of their legacy code bases. Hence, they also served as an *in-vivo* testing environment for the project.

In the rest of the paper we give an overview of the features and the architecture of FaultBuster and we show a typical usage scenario of the tools. Then we present a comparison of our tool with similar tools that are available for automatic refactoring.

II. Overview

A. Problem Context

The potential users of FaultBuster are members of a development team, potentially a developer or maybe a quality

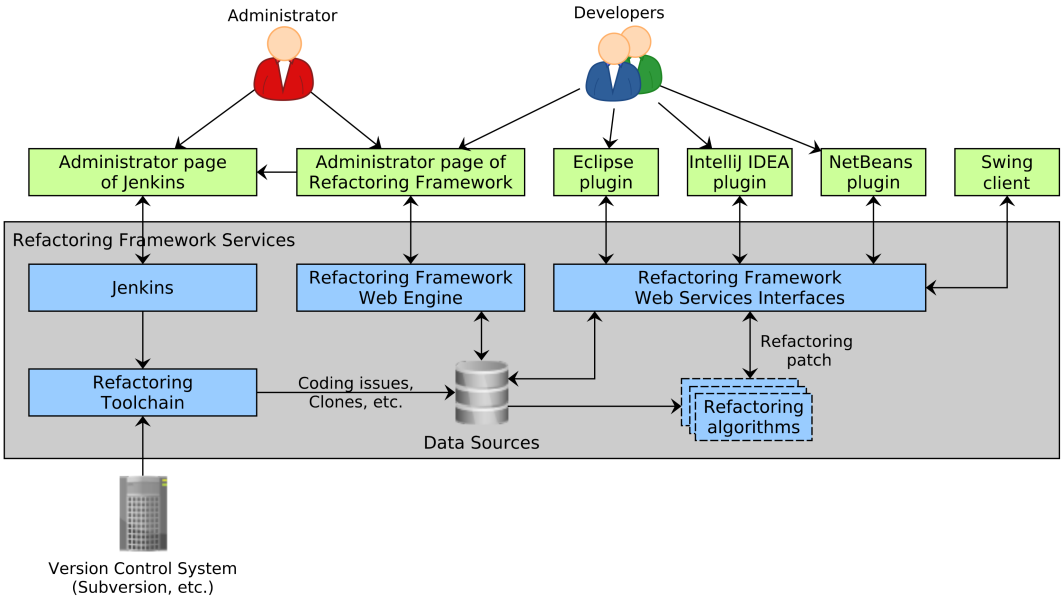

Figure 1. Overview of the architecture of FaultBuster

specialist or a lead developer. Our purpose was to help them by supporting '*continuous refactoring*' where developers prefer to make small improvements on a regular basis instead of only adding new features for a longer period and restructuring the whole code base only when real problems arise. Continuous refactoring has many benefits [5], as Kerievsky says "*by continuously improving the design of code, we make it easier and easier to work with. This is in sharp contrast to what typically happens: little refactoring and a great deal of attention paid to expediently adding new features. If you get into the hygienic habit of refactoring continuously, you'll find that it is easier to extend and maintain code*" [6].

For this purpose, FaultBuster was designed to periodically analyze the system under question, report problematic code fragments and provide support to fix the identified problems through automatic transformations (refactorings).

We notice here, that most of the transformations supported by FaultBuster can be considered as classic refactorings which do not alter the external behavior at all, just improve the internal structure of the source code, but some of them may not fit into the classic definition. As Fowler says, "*refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure*" [2]. Some of the transformations fix potential bugs, which may indeed alter the behavior of the program, but not the behavior which was originally intended to be implemented by the developer. Hence, for some transformations, it means that we slightly deviate from the strict definition by allowing changes to the code that fix possible bugs, but do not alter the behavior of the code in any other way. For simplicity, we call refactorings all the transformations of FaultBuster.

A sample refactoring of a coding rule violation (*Position Literals First In Comparisons*) can be seen in Listing 1. This code works perfectly, until we invoke the 'printTest' method with a null reference which would result in a *Null Pointer Exception* (because of line 3). To avoid this, we have to compare the String literal to the variable, not the variable to the literal (see Listing 2). This and similar refactorings are simple, but one can avoid critical or even blocker errors using them properly.

```
public class MyClass{
 public static void printTest(String a){
  if(a.equals("Test")) {
   System.out.println("This is a test!");
  }
 }
 public static void main(String[] args) {
  String a = "Test";
  printTest(a);
  a = null;
  printTest(a); // What happens?
 }
}
```

Listing 1. A sample 'Position Literals First In Comparisons' issue

```
public class MyClass{
 public static void printTest(String a){
  if("Test".equals(a)) {
   System.out.println("This is a test!");
  }
 }
 public static void main(String[] args) {
  String a = "Test";
  printTest(a);
  a = null;
  printTest(a); // What happens?
 }
}
```

Listing 2. Sample refactoring of Listing 1

B. Architecture

Figure 1 gives an overview of the architecture of FaultBuster. The toolset consists of a core component called *Refactoring Framework*, three IDE plugins to communicate with the framework and a standalone Java Swing client (desktop application).

1) Refactoring Framework: This component is the heart of FaultBuster as its main task is to control the whole refactoring process. The framework deals with the continuous quality measurements of the source code, identification of critical parts from the viewpoint of refactoring, the restructuring of them, the measurement of quality improvement and the support of regression tests to verify the invariance after applying the refactorings.

In order to do so, the framework:

- Controls the analysis process and stores the results in a central database: it regularly checks out the source code of the system from a version control system (Subversion, CVS, Mercurial, Git), executes static analyzers (Java analyzer, rule checker, code smell detector, etc.) and uploads the results into the database.
- Provides an interface through web services to query the results of the analyses and to execute automatic refactoring algorithms for selected problems. After executing the algorithms on server side, the framework generates a patch (diff file) and sends it back to the client.
- The analysis toolchain is controlled and can be configured through Jenkins.
- Refactoring algorithms and main settings of the framework are configurable through a web engine of the framework.

The framework was designed to be independent from the programming language, so it is suitable to support new languages and to be easily extensible with additional refactorings. Several modules are integrated for the realization of the task: well-known tools supporting development procedures, like version control systems, project management tools, development environments, tools supporting tests, tools measuring and qualifying source code and automatic algorithms implementing refactorings.

2) IDE plugins: We implemented plugins for today's most popular development environments for Java (Eclipse, NetBeans, IntelliJ IDEA) and integrated them with the framework. The goal of these plugins is to bring the refactoring activities to be implemented closer to the developers.

A plugin obtains a list of problems in the source code from the framework, processes the results, and shows the critical points which influence software quality negatively to the user. A developer can then select one or more problems from this list and ask for solution(s) from the framework, which can then be visualized and (after confirmation) applied to the code by the plugin. Finally, the developer can make some minor changes to it (e.g. commenting) and commit the final patch to the version control system.

When we designed the plugins, the main concept was to integrate the features offered by the framework as much as we can into the development environment. For example, we implemented standard features such as *context assist* in Eclipse. So it was a main concern that developers can work in the environment that they are used to and access the new features in a standard way.

Figure 2 shows a screenshot of the Eclipse plugin with a wizard to set parameters of an algorithm which fixes a Long Function issue. Figure 3 shows the visualization of a patch after the execution of the algorithm.

3) Standalone Swing Client: Besides the IDE plugins, we implemented a standalone desktop client to communicate with the Refactoring Framework. In the beginning this client had only testing purposes, but finally it implemented all the necessary features of the whole system, so it became a useful standalone tool of FaultBuster. The client is able to browse the reports on problematic code fragments in the system, select problems for refactoring, and invoke the refactoring algorithms, just like IDE plugins can do.

4) Administrator Pages: The framework has two graphical user interfaces to configure its settings. Analysis tasks are controlled by Jenkins to periodically check out the source code and to execute the static analyzers. These tasks can be configured through the admin page of Jenkins. The rest of the framework can be configured through its own admin pages. Here, it is possible to configure user profiles and set some global parameters of the refactoring algorithms. In addition, this UI can be used to examine log messages and statistics of the framework.

5) Refactoring Algorithms: We implemented automatic refactoring algorithms to fix common code smells and bad programming practices. The input of such an algorithm is a coding issue (with its kind and position information) and the abstract semantic graph (ASG) of the source code generated by the SourceMeter tool (see Section II-C). The output of an algorithm is a patch (unified diff file) fixing the selected problem.

FaultBuster implements algorithms to solve 40 different kinds of coding issues (see Table I) in Java. Most of these algorithms solve common programming flaws like 'empty catch block', 'avoid print stack trace', 'boolean instantiation,' while some of them implement heuristics to fix bad code smells such as long function, too complex methods or code duplications.

Some algorithms can interact with the developer, because they can be parametrized. For instance, in the case of a 'method naming convention' issue it is possible to ask the developer to give a new name for the badly named method. On the other hand, many algorithms do not need extra information, e.g. the case of a 'local variable could be final' issue, the final keyword can be simply inserted to the declaration of the variable automatically.

It is also possible to select more occurrences of the same problem type and fix them all together by invoking a so-called batch refactoring task. In this case, the Refactoring Framework

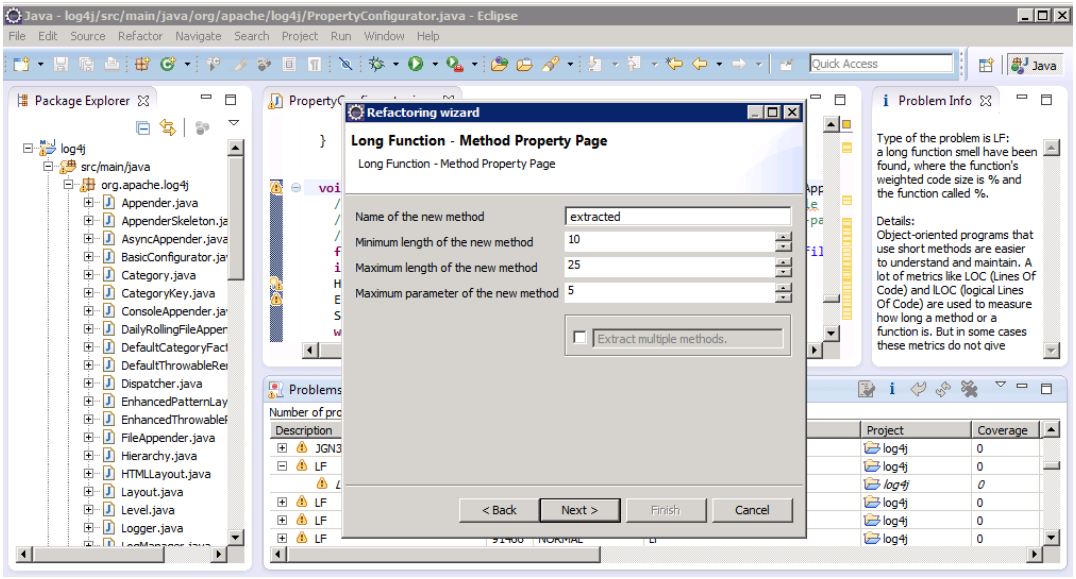

Figure 2. Eclipse plugin – Screenshot of a Refactoring wizard with the configuration step of a refactoring algorithm for the Long Function smell

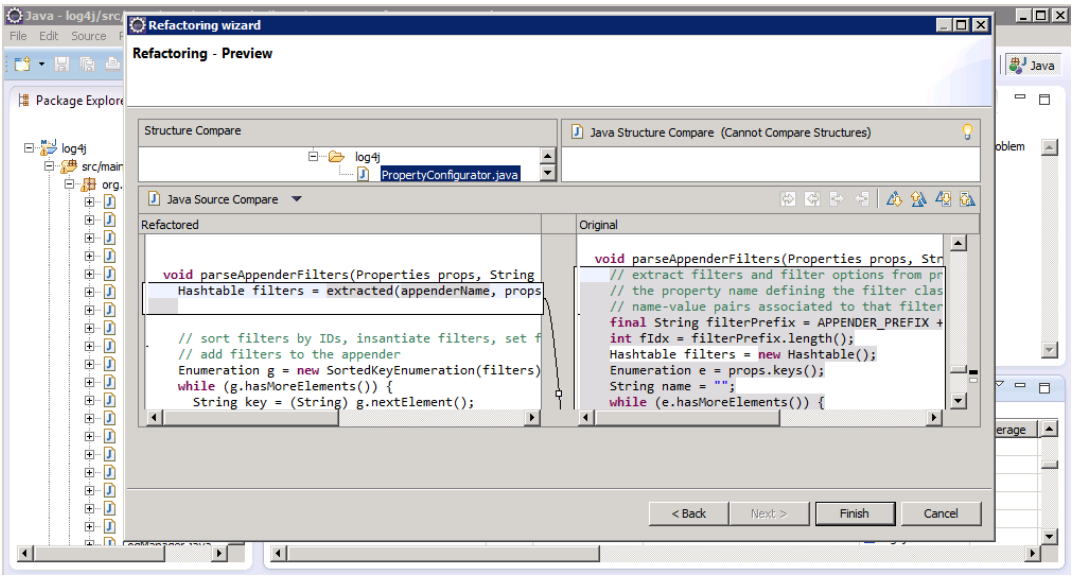

Figure 3. Eclipse plugin – Difference view of a patch after refactoring a Long Function smell

will execute the refactoring algorithms and will generate a patch containing the fixes for all the selected issues. The only limit here is the boundary of the analysis, so it is possible to select problems from any classes, packages or projects, they just had to be analyzed beforehand by the framework.

C. Technologies

The Refactoring Framework of FaultBuster relies on the SourceMeter product[1] of FrontEndART Ltd.[2] as a toolchain for static analysis and to generate the ASG for the refactoring algorithms. The core framework was implemented in Java as a Tomcat Web Application and serves the IDE plugins through web services. Refactoring algorithms were implemented in Java using the Columbus ASG API of SourceMeter. Thanks to the Tomcat environment the toolset is platform-independent and runs on all the supported platforms of SourceMeter (Windows and Linux).

[1]http://sourcemeter.com
[2]http://frontendart.com

III. Experiences

Thanks to the project which motivated the development of FaultBuster, we had chance to get immediate feedback from potential users of the tool in all stages of its development (starting from the design phases to the last testing phases of the project). We gathered a lot of experience in how to design

Table I
REFACTORING ALGORITHMS IN FAULTBUSTER

Local	AddEmptyString
	ArrayIsStoredDirectly
	AvoidReassigningParameters
	BooleanInstantiation
	EmptyIfStmt
	LocalVariableCouldBeFinal
	PositionLiteralsFirstInComparisons
	UnnecessaryConstructor
	UnnecessaryLocalBeforeReturn
	UnusedImports
	UnusedLocalVariable
	UnusedPrivateField
	UnusedPrivateMethod
	UselessParentheses
Naming	BooleanGetMethodName
	MethodNamingConventions
	MethodWithSameNameAsEnclosingClass
	ShortMethodName
	SuspiciousHashcodeMethodName
Interactive	AvoidInstanceofChecksInCatchClause
	AvoidPrintStackTrace
	AvoidThrowingNullPointerException
	AvoidThrowingRawExceptionTypes
	EmptyCatchBlock
	LooseCoupling
	PreserveStackTrace
	ReplaceHashtableWithMap
	ReplaceVectorWithList
	SimpleDateFormatNeedsLocale
	SwitchStmtsShouldHaveDefault
	UseArrayListInsteadOfVector
	UseEqualsToCompareStrings
	UseLocaleWithCaseConversions
	UseStringBufferForStringAppends
Heuristical	Clone Class (experimental)
	CyclomaticComplexity
	ExcessiveMethodLength
	LongFunction
	NPathComplexity
	TooManyMethods

and implement such a tool, and also on the final usability of FaultBuster, which we briefly sum up in this section.

During the design phase of the tool we consulted regularly with the developers of the participating companies about the refactoring transformations which they wanted to be available in the final product. Throughout the initial meetings it became clear that they wanted ready solutions for their actual problems, particularly for those which were easily understandable for the developers and by solving them, they could gain the most in terms of increasing the maintainability of their products. However, they did not really provide us a concrete list of the issues that they wanted to deal with. In addition, most of the developers said that before the project they had not used any refactoring tools except the ones provided by their IDEs. Therefore, we started implementing transformation algorithms to fix coding rule violations which were very common in their projects. Soon, when we provided the companies the first prototype versions for testing, they started to send us tons of other issue types and refactoring algorithms that they wanted to be supported in the new releases. Among the desired refactorings, there were some more complex ones too like eliminating long or complex methods. In the end, we also implemented an algorithm which eliminates clones (code duplications) from the source code.

At the end of the project, we can say, that FaultBuster performed well and was tested exhaustively by the companies. The companies participating in the project performed around 6,000 refactorings altogether which fixed over 11,000 coding issues. Interviews with the developers showed that they found the work with the tool really helpful in many situations and they intend to keep using it in the future.

IV. RELATED TOOLS

Since the introduction of the term 'refactoring' [1], many researchers studied it [7] as a technique, e.g., to improve source code quality [8], [9], and many tools were implemented providing different features to assist developers in refactoring tasks. FaultBuster is a refactoring tool to detect and remove code smells, i.e., in this section, we give an overview of tools which have similar capabilities, or are related to FaultBuster through some specific features.

In a recent study, Fontana et al. examined refactoring tools to remove code smells [4]. They evaluated the following tools: Eclipse,[3] IntelliJ IDEA,[4] JDeodorant,[5] and RefactorIT.[6] In the case of JDeodorant, they say that this „*is the only software currently available able to provide code smell detection and then to suggest which refactoring to apply to remove the detected smells.*" To evaluate the other refactoring tools, they relied on the code smell identification of iPlasma[7] and inCode[8].

In an earlier study, Pérez et al. also identified smell detection and automatic corrections as an open challenge for the community, and proposed an automated bad smell correction technique based on the generation of refactoring plans [10].

On the side of detecting bad smells, there are many static analyzers to automatically identify programming flaws, like the products of Klocwork Inc.[9] or Coverity Inc.[10] These tools are sometimes able to identify serious programming flaws (e.g. buffer overflow or memory leak problems) that might

[3]https://www.eclipse.org/
[4]http://www.jetbrains.com/idea/
[5]http://www.jdeodorant.com/
[6]http://sourceforge.net/projects/refactorit/
[7]http://loose.upt.ro/reengineering/research/iplasma
[8]http://www.intooitus.com/products/incode
[9]http://www.klocwork.com/
[10]http://www.coverity.com/

lead to critical or blocker problems in the system. There are open source or free solutions as well, such as PMD,[11] FindBugs,[12] CheckStyle,[13] for Java, or the Code Analysis features and FxCop in Visual Studio.[14] These tools usually implement algorithms to detect programming flaws, but fixing the identified issues remains the task of the developers.

The DMS Software Reengineering Toolkit[15] product of Semantic Designs Inc. has a 'program transformation engine' which enables the tool to do code generation and optimization, and makes it able to remove duplicated code (with CloneDR).

There are many IDEs available with automatic refactoring capabilities and they support typical code restructurings (e.g. renaming variables, classes) and some common refactorings from the Fowler catalog. For instance, IntelliJ IDEA was one of the first IDEs to implement these techniques and it is able to support many languages (e.g. PHP, JavaScript, Python), not only Java which it was originally designed for. Eclipse and NetBeans also implement similar algorithms. However, neither of these IDEs support the automatic refactoring of programming flaws. On the other hand, there are many plugins available to extend their refactoring capabilities, such as ReSharper[16] and CodeRush[17] for .NET.

Compared to these tools, they all lack the feature of scanning the code and suggesting which refactorings to perform, which is one of the main advantages of FaultBuster. JDeodorant, as an Eclipse plug-in, is the only tool, which has a similar capability, as it is able to identify four kinds of bad smells (namely 'Feature Envy', 'State Checking', 'Long Method' and 'God Class'), and refactor them by the combination of 5 automatic refactoring algorithms. FaultBuster is more general in a way, as it allows the refactoring of coding issues (see Table I) and has plug-in support for IntelliJ and NetBeans too (besides Eclipse).

Another main feature of FaultBuster is the ability to effectively perform a large set of refactorings (i.e. batch refactorings) together on a large code base. The lack of tools which are able to handle a massive Java code base and to provide a large class of useful refactorings also motivated the development of refactoring tools, e.g. Refaster[18] from Google [11].

V. Conclusion

FaultBuster was implemented as a product of an R&D project supported by the EU and the Hungarian Government. Six companies were involved in this project and in addition to the project goal (to implement automatic refactoring tools) they wanted to improve the source code of their own products as well. Hence, they provided a real-world test environment and they tested the tools on their own products. Thanks to this context, the implementation of the toolset was driven by real, industrial motivation and all the features and refactoring algorithms were designed to fulfill the requirements of the participating companies. We implemented refactoring algorithms for 40 different coding issues, mostly for common programming flaws. By the end of the project the companies refactored their systems with over 5 million lines of code in total and fixed over 11,000 coding issues. FaultBuster gave a complex and complete solution for them to improve the quality of their products and to implement continuous refactoring to their development processes.

We have many plans to improve FaultBuster for future work. Primarily we want to extend it with more refactoring algorithms, to support different kinds of coding issues. However, it is important to see that an algorithm can only be effective if the method which identifies the coding issue is effective too. Although we implemented our refactoring algorithms with preliminary checks where we decide whether a selected coding issue can be fixed or not, our partners reported also some false positive warnings. To eliminate these we plan to improve the precision of the underlying issue checker too.

A FaultBuster demo can be downloaded and a video can be found in the online appendix of this paper at:
http://www.sed.inf.u-szeged.hu/FaultBuster.

[11] http://pmd.sourceforge.net/
[12] http://findbugs.sourceforge.net/
[13] http://checkstyle.sourceforge.net/
[14] https://msdn.microsoft.com/en-us/library/bb429476(v=vs.80).aspx
[15] http://www.semdesigns.com/products/DMS/DMSToolkit.html
[16] http://www.jetbrains.com/resharper/
[17] https://www.devexpress.com/Products/CodeRush/
[18] https://github.com/google/Refaster

Acknowledgements

This research work was supported by the EU supported Hungarian national grant GOP-1.2.1-11-2011-0002. Here, we would like to thank all the participants of this project for their help and cooperation.

References

[1] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, 1992.
[2] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999.
[3] G. H. Pinto and F. Kamei, "What programmers say about refactoring tools?: An empirical investigation of Stack Overflow," in *Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools*, ser. WRT '13. ACM, 2013, pp. 33–36.
[4] F. A. Fontana, M. Mangiacavalli, D. Pochiero, and M. Zanoni, "On experimenting refactoring tools to remove code smells," in *Scientific Workshop Proceedings of the XP2015*, ser. XP '15 workshops. New York, NY, USA: ACM, 2015, pp. 7:1–7:8.
[5] T. Bakota, P. Hegedus, G. Ladanyi, P. Kortvelyesi, R. Ferenc, and T. Gyimothy, "A cost model based on software maintainability," in *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM)*, Sept 2012, pp. 316–325.
[6] J. Kerievsky, *Refactoring to Patterns*. Pearson Higher Education, 2004.
[7] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Trans. Softw. Eng.*, vol. 30, no. 2, pp. 126–139, Feb. 2004.
[8] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.
[9] B. Du Bois, "A study of quality improvements by refactoring," Ph.D. dissertation, 2006.
[10] J. Pérez and Y. Crespo, "Perspectives on automated correction of bad smells," in *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, ser. IWPSE-Evol '09. New York, NY, USA: ACM, 2009, pp. 99–108.
[11] L. Wasserman, "Scalable, example-based refactorings with Refaster," in *Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools*, ser. WRT '13. New York, NY, USA: ACM, 2013, pp. 25–28.