Designing and Developing Automated Refactoring Transformations: An Experience Report

Gábor Szőke*†, Csaba Nagy*†, Rudolf Ferenc* and Tibor Gyimóthy*
{gabor.szoke, ncsaba, ferenc, gyimothy}@inf.u-szeged.hu
*Department of Software Engineering, University of Szeged, Hungary
†Refactoring 2011 Ltd., Hungary

***Abstract*—There are several challenges which should be kept in mind during the design and development phases of a refactoring tool, and one is that developers have several expectations that are quite hard to satisfy. In this report, we present our experiences of a two-year project where we attempted to create an automatic refactoring tool. In this project, we worked with five software development companies that wanted to improve the maintainability of their products. The project was designed to take into account the expectations of the developers of these companies and consisted of three main stages: a manual refactoring phase, a tool building phase, and an automatic refactoring phase. Throughout these stages we collected the opinions of the developers and faced several challenges on how to automate refactoring transformations, which we present and summarize.**

***Index Terms*—Automated Refactoring; Code Smells; Coding Issues; Software Maintenance**

I. Introduction

Several papers investigate the benefits and challenges of refactoring [1], e.g., its effects on software quality or maintainability [2]–[8], but there are only a few reports that tackle this topic from the developers' viewpoint. Recently, Kim et al. [9] conducted a field study of refactoring definition, benefits, and challenges. They surveyed developers of a large software development company and pointed out that even the refactoring definition in practice seems to differ from the academic definition. Pinto et al. [10] examined Stack Overflow posts to investigate what developers expect from refactoring tools, and they found that most of these expectations are far from being complete. Developers may overlook several features, for example, they may miss refactoring recommendations like the identification of duplicate or dead code, code smells, and optimization opportunities. In a workshop paper, Campbell et al. [11] say that common shortcomings of refactoring tools are that developers find them difficult to learn, they do not trust the tool, or they are not familiar how to use them. Also, developers often feel that they can apply refactorings more efficiently by hand.

We also observed similar findings in a project that motivated our experience report. In our project, we worked together with five companies to develop automatic refactoring tools that could be used to improve the source code quality of their systems. This project was meant to take into account the requirements of the developers of these companies, and we ended up facing several challenges to fulfill all the expectations of the developers and companies.

The project lasted for two years and it had three main stages: *analysis* as a manual refactoring stage, *design & development*, and *application* as an automatic refactoring stage. A key idea of our project plan was to collect the opinions of the developers in each phase, so the next phase could build on the previous one and take into account the feedback of the developers.

In previous studies, we investigated how refactoring transformations affected the source code maintainability when the developers manually applied them on the code [12], [13]. We found that the effect of a single transformation usually has an unbalanced positive/negative effect on one or more quality attributes, but when developers applied them *en masse*, it always had a beneficial effect on the maintainability of the code. Later, in the automatic refactoring phase, we found that developers tend to have a different behavior when they use automatic tools (e.g., they accept a solution recommended by the tool even if manually they would probably do something else), so we examined the automatic transformations as well [14]. In a recent tool demo paper [15], we described the architecture of the tools developed in the project.

Here, we report our experiences and the challenges that we faced while we designed and implemented the tools, but unlike earlier reports [10], [11], we focus on the automatic transformations and not on the general usability of the tool. Also, developers provided us with several recommendations in the manual phase (How did they refactor? Do they think that it is possible to automate their steps? If yes, how would they automate them?). Then, they gave us feedback on the resulting implementations. Besides our experiences, we also examined their opinions. Therefore, our report can serve as a guideline for others, who face similar challenges.

II. Refactoring Project

A. Project background

Our motivation was a two-year R&D project supported by the EU and the Hungarian State. One goal of the project was to develop software tools to support the '*continuous reengineering*' methodology, hence provide support to identify problematic code parts in a system and to refactor them to enhance maintainability. This included the development of an automatic refactoring framework and the testing of it on the source code of the industrial partners. Hence, we had an *in vivo* environment and continuous feedback on the tools. Moreover, the project provided the companies with a good opportunity to refactor their code and improve its maintainability.

Table I
COMPANIES INVOLVED IN THE PROJECT

Company	**Primary domain**
Company I	Enterprise Resource Planning (ERP)
Company II	Integrated Business Management
Company III	Integrated Collection Management
Company IV	Specific Business Solutions
Company V	Web-based PDF Generation

Five experienced software companies were involved in this project. They were founded in the last two decades, and they started developing some of their systems before the millennium. The systems that they refactored in the project consisted of about 2.5 million lines of code altogether, had been written mostly in Java, and were related to different areas like ERPs, ICMs, and online PDF Generators (see Table I).

B. Project design

Figure 1 gives an overview of the main stages of the project. In the first stage (*Analysis*), we asked the companies to refactor their code manually. We gave them support by using static code analyzers to help them identify code parts that should be refactored in their code (antipatterns or coding issues, for instance). We asked the developers to provide detailed documentation of each refactoring, and explain the main reasons and the steps of how they improved the code fragment in question.

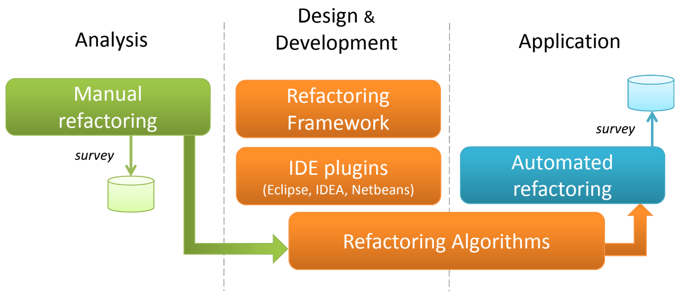

Figure 1. Overview of the refactoring project.

In the second stage (*Design & Development*), we designed and implemented a refactoring framework [15] based on the results of the manual refactorings. This framework was implemented as a server-side component that provided three types of services:

- A static source code analyzer toolset to derive low-level quality indicators that could be used to identify refactoring candidates.
- A persistence layer above a database for storing and querying analysis data (with a complete history).
- A set of web services capable of automatically performing various refactoring operations to eliminate certain coding issues and generate a source code patch to be applied on the original code base.

As can be seen from the above list, the framework not only provided refactoring algorithms for the developers, but it also helped to identify possible targets for refactoring by analyzing their systems using a static source code analyzer. The tool was able to give a list of problematic code fragments including coding issues, antipatterns (e.g. duplicated code, long functions) and source code elements with problematic metrics at different levels (e.g. classes/methods with excessive complexity and classes with bad coupling or cohesion metrics). However, the framework only supported the refactoring of 40 different coding issues, so the companies were just asked to fix issues from this list.

The participating companies took part in the development of the refactoring tools as well. One of their tasks was to develop IDE plugins for their own working environments (Eclipse, IDEA, and Netbeans). So it was the responsibility of the framework to perform the refactoring transformations and generate patches. The IDE plugins were responsible for providing an interface to all the features of the framework by taking advantage of the UI elements of the IDEs. This way, the refactoring process was controlled by the framework and the developers worked in their familiar workspace.

In the third stage of the project (*Application*), the developers used the automatic tool to refactor their code base. Over 7,800 issues got fixed, which fell into about 30 different kinds of issues. Thanks to the project requirements, all the refactorings were well documented.

III. WHAT DEVELOPERS THINK ABOUT REFACTORING AUTOMATION?

Throughout the manual refactoring and the automatic refactoring phases, we asked developers to fill out surveys for the refactoring operations they had carried out. For each refactoring commit, they had to fill out a survey that contained questions targeting the initial identification steps, and they also had to explain why, how, and what they modified in their code. There were around 40 developers involved in this phase of the project (5-10 per company). The questions related to our study were the following:

- *How difficult would it be to automate your **manual refactoring** for the issue?* (1 - very easy, 5 - very hard) + explanation
- *How much did the **automated refactoring** help in your task?* (1 - no help at all, 5 - great help) + explanation

A. Manual refactorings

During the manual refactoring phase of the project, developers refactored their codebase manually, and they filled out a survey for each refactoring. We had an online Trac system for this purpose, and whenever they opened a ticket for an issue, they had to explain why they found it problematic, and answer some additional questions. Similarly, we asked them some questions when they closed the ticket after they had finished the refactoring.

Among these questions, they had to rate with a value from 1 to 5 (1 - very easy, 5 - very hard) how difficult it would be to automate the manual refactoring. Along with this number, they had to give a brief explanation of their answer.

The developers completed the survey for 430 tickets, as can be seen in Table II. Our results tell us that developers gave responses for 61 different kinds of coding issues (actually we

Table II
DEVELOPERS' FEEDBACK ON HOW HARD IT WOULD BE TO AUTOMATE REFACTORING OPERATIONS

Num. of Replies	Avg	Med	Dev	Num. of Types
430	2.06	1	1.23	61

showed them around 220 different kinds of coding issues). Figure 2 shows the histogram of the given replies. As can be seen, most of the refactorings were rated with smaller values, which indicates that they were optimistic about the automation: they thought that most of the coding issues could be easily fixed through automated transformations. However, they also identified some cases where they thought that the automation would be hard to realize. Notice also that Table II also supports this observation, as the average value is around 2 and the median is 1. Table III lists the coding issues and the level of difficulty of their automation based on feedback of the developers.

Table III
HOW DIFFICULT THE REFACTORING AUTOMATION OF CODING ISSUES IS ACCORDING TO DEVELOPERS

Aut.	Coding Issues
Very Hard (5)	AvoidInstanceofChecksInCatchClause, ExceptionAsFlowControl, EmptyIfStmt
Hard (4)	SwitchStmtsShouldHaveDefault, AvoidCatchingThrowable, MethodReturnsInternalArray
Medium (3)	UseStringBufferForStringAppends, AvoidSynchronizedAtMethodLevel, SignatureDeclareThrowsException, AvoidCatchingNPE, AbstractClassWithoutAbstractMethod, ConsecutiveLiteralAppends, LooseCoupling, NonThreadSafeSingleton, ReplaceHashtableWithMap, SystemPrintln, UnusedFormalParameter, UseLocaleWithCaseConversions, UnsynchronizedStaticDateFormatter
Easy (2)	EmptyCatchBlock, OverrideBothEqualsAndHashcode, PreserveStackTrace, UnnecessaryLocalBeforeReturn, AtLeastOneConstructor, UnusedPrivateField, UnusedPrivateMethod, AvoidThrowingRawExceptionTypes, UnusedLocalVariable, AvoidDuplicateLiterals, AvoidDeeplyNestedIfStmts, AddEmptyString, AvoidFieldNameMatchingTypeName, ArrayIsStoredDirectly, AbstractNaming, ImmutableField, OnlyOneReturn, UnnecessaryConstructor, UnnecessaryWrapperObjectCreation
Very Easy (1)	AvoidPrintStackTrace, UnusedImports, UseIndexOfChar, InefficientStringBuffering, IntegerInstantiation, MethodArgumentCouldBeFinal, CyclomaticComplexity, BooleanInstantiation, BigIntegerInstantiation, BeanMembersShouldSerialize, CollapsibleIfStatements, CompareObjectsWithEquals, IfElseStmtsMustUseBraces, LocalVariableCouldBeChar, SimplifyConditional, ShortVariable, UncommentedEmptyMethod, UnnecessaryFinalModifier, UnusedModifier, UnnecessaryReturn, VariableNamingConventions

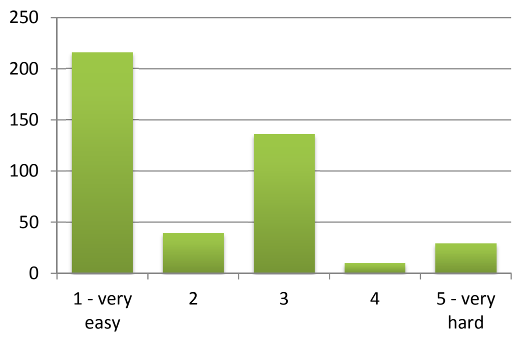

Figure 2. Histogram of the answers given for "*How difficult would it be to automate your manual refactoring for the issue?"*

B. Automated refactorings

We started implementing the automated refactorings based on our observations got in the manual phase. Also, we asked the companies to provide us with a list of coding issues (with priorities) that they wanted to fix in the automated phase so that we could concentrate on the most desired ones. After gathering the lists, we ranked each coding issue by the values the companies provided us. Then we created a ranked list of the coding issues that most of the companies wanted at the top, and the coding issues that nobody wanted at the end. Interestingly, the resulting list contained many issues that were no longer considered during the manual phase, most probably because the companies fixed all occurrences of some issue types so they were not interested in the automation of these.

We started implementing the refactoring algorithms based on this ordered list. We developed automatic refactoring solutions for 42 different coding issues. The supported list of coding issues consisted of 22 different issue types that were considered during the manual period plus 20 new ones.

During the automatic refactoring stage, we asked the developers to document their refactorings again. This time, we incorporated the survey into our tool that asked them to fill it out after each refactoring transformation. This way, we gathered over 1,700 answers for 30 coding issue types (see Table IV).

Table IV
TOTAL HELP FACTOR SURVEY

Num. of Replies	Avg	Med	Dev	Num. of Types
1,726	3.05	3	1.23	31

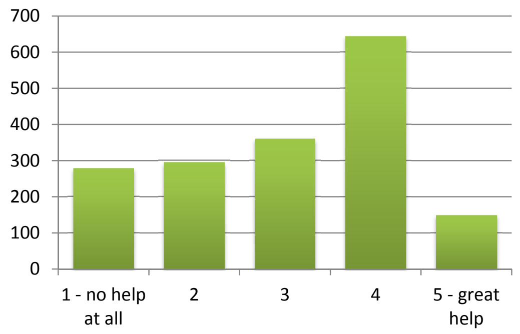

Figure 3. Histogram of the answers given for the question *"How much did the automated refactoring help in your task?"*

In the automatic phase, we asked developers about how much the automated refactoring solution aided them in their refactoring task. They had to give a value between 1 and 5 here as well. A 5 meant that the automation helped a lot, while a 1 meant that it did not help at all (or it even made the situation worse). As we see in Table IV, the average of the replies was around three. In Figure 3, the distribution of the responses can be seen in a histogram. This tells us that developers were generally satisfied with the automated refactoring solutions, and they gave a score of 4 in many cases.

Actually, if we consider all the transformations where the given value is greater than 1 (these are the transformations that the developers said were a help), we find that all the refactorings made the tasks of developers easier or faster, except for two cases. This can be seen in Figure 4, where we can see the degree of help for each kind of coding issue.

The points stand for the average help of a refactoring solution and the bars around them indicate the standard deviation.

Every refactoring algorithm for a coding issue got a value above 1, except the *LooseCoupling* and the *MethodNaming-Convetions* coding issues. In their explanations, the developers said that they found that fixing one issue had a communication overhead that sometimes made it easier for them to refactor the code manually in the IDE instead. However, this overhead might be negligible if they fix more issues together. For example, the refactoring solution for *MethodNamingConvetions* issue suggests a better name for a method (e.g. if a method name starts with an uppercase letter it recommends the same name beginning with a lowercase letter). After the developer accepted the refactoring suggestion, they had to wait until our tool applied the modification. This could take a few seconds because of the server-client architecture.

Upon examining Figure 4 again, we realized that when we consider not only the average but also the standard deviation for each coding issue, we can classify the following 5 refactoring types as 'sometimes bad': *LongFunction*, *Cyclomatic-Complexity*, *UseStringBufferForStringAppends*, *UselessParentheses*, *TooManyMethods*.

The developers explained these as follows. The *Useless-Parentheses* issue fell into the same category as the former two; it is faster to do it manually in some cases. The *Long-Function* and *CyclomaticComplexity* issue fixing refactoring solutions used an extract method refactoring algorithm where the algorithm applied a heuristic to find parts of the code that can be extracted to satisfy the requirement by the issue, to reduce the length of the method or to reduce the complexity of it. The main problem with this algorithm was that it was hard for developers fathom how it worked. They simply preferred to do it manually instead of using the tools. The *TooManyMethods* issue suffered from the same problem, but in this case the underlying algorithm was 'extract class'. Developers' notes on the *UseStringBufferForStringAppends* issue show that although they were satisfied with the semantic aspect of the algorithm, many formatting problems arose.

IV. Experiences

Throughout the lifetime of the project we faced many challenges. Here, we present our experiences that we learned from the feedbacks of the developers and from their responses given to the survey.

A. Challenges in how to automate refactoring transformations

1) Precise syntax tree: Without a doubt, a key considerations of refactoring transformations is to *have a precise representation of the source code*. One can model the source code as an *abstract syntax tree* (AST) to perform different (graph) transformations on it. Transformations can be just as good as the underlying representation is, so we found it necessary to have an accurate and complete AST. To illustrate this, consider a rename method refactoring. Here, we do not simply change the method name, but *a*) we have to check that the new name does not conflict with other method names in the same scope (e.g. parent and child classes); *b*) we have to check for disambiguation in other classes where the method is invoked; *c*) and then, when it passes the former two checks we are allowed to rename the method and all of its invocations to the new name. To do this, we have to analyze all the dependencies, and potentially include external ones as well.

2) Regenerate (only the) modified code: After the transformations on the AST, we have to apply the changes to the source code. To do this, we have to (re)generate the source code from the AST (at least, and preferably only for the modified code parts). It is also important not to introduce unnecessary changes to the other parts of the source code.

3) Code formatting: The process of code generation requires some indentation and code formatting as well. It is usually hard for the users to specify formatting rules, and hence it is also hard to regenerate a code formatted exactly as the user would like to see it. This was one of the most difficult challenges we could not fully overcome, and this caused the most dissatisfaction among developers. However, based on our experiences, developers mostly accepted this limitation if they found the refactoring to be semantically correct and they had to reformat the code only a bit manually (e.g., they could easily do that in the IDE automatically).

4) Patch generation: As the last step after the transformation, we generated a *diff* (difference file or patch) between the old source code and the new one. Then we sent this diff file to the IDE where the developers could decide to accept or reject the modification.

5) Code clone elimination: Another interesting experience was that the developers eagerly wanted to eliminate code duplications (code clones). By the end of the project, we developed an experimental algorithm that was able to refactor code clones via extract method and extract class refactoring transformations. Note that automated code clone detection is a hot research topic as well [16], especially code clone elimination. It is quite a challenge to come up with a solution for this issue.

B. What makes a refactoring operation good or bad?

1) Precise problem detection: Developers only wanted recommendations made for real faults or optimization opportunities, and they wanted to avoid false positives. Looking at false recommendations takes time, and it does not bring any benefit to the project. Besides false positive issues, they also wished to avoid true negative issues. As a common use case, they said they wanted to remove all the occurrences of a certain type of issue. Reporting only some occurrences would give them a false sense of security.

2) Understandability of the transformations: Refactorings with a good and easy-to-comprehend description were more popular among the developers. Unlike those refactoring solutions that required more parameters or were harder to understand, developers rarely used these and gave worse scores in the survey.

3) Performance: It was important to carry out the modifications quickly, or at least quicker than it would be to do manually.

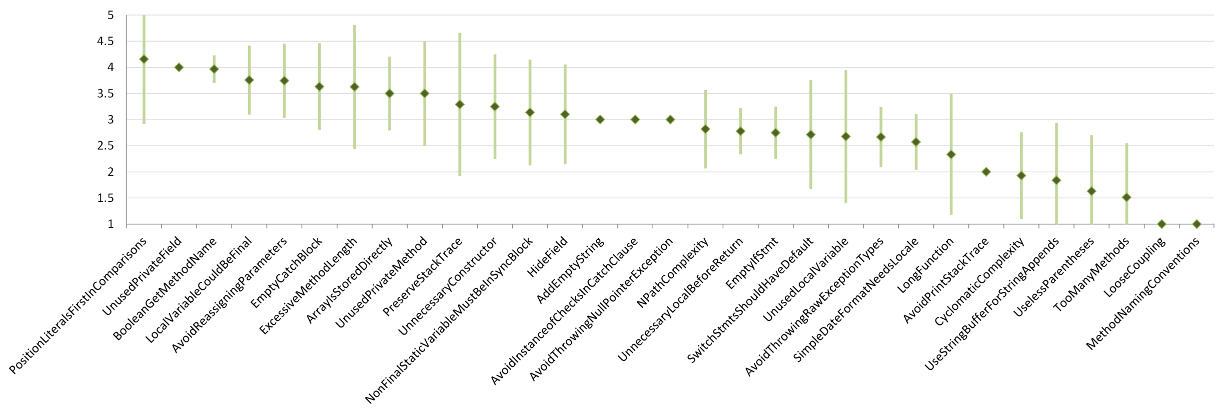

Figure 4. How much the automated refactoring solution assisted the developers (5 - great help, 1 - no help at all)

4) Batch refactorings: One way to improve efficiency is by supporting the refactoring of several issues at the same time. With the automated tool, developers were able to fix many issues of the same type all at once (we called this *batch* refactoring). This batch-refactoring process made refactoring tasks a lot faster. For example, they were able to fix all occurrences of *UnusedConstructor* issues with a press of a button. This option was beloved by developers, and batch refactorings got better scores in the survey. However, we did not allow batch refactoring of all the issues. We had to implement some restrictions in this process because we observed that developers tended to accept these refactorings without checking the result of the automated refactoring operations. This was flattering because it meant that they trusted the algorithm and its results. Nonetheless, we did not want them to blindly accept the refactorings. Therefore, we only allowed the refactoring of one type of issue at a time, and we only allowed it for some simpler refactorings. This way we guaranteed that they check complex refactorings (e.g. extract class) and ensured that simpler ones run fast.

5) Comment handling: Comments are integral parts of the source code, and sometimes they are closely related to source code elements. Developers expected from the transformations that they would also handle these situations. For instance, a refactoring that removes an unused constructor should remove the comment before the constructor as well. Similarly, in some cases they asked us to generate simple comments.

V. Conclusions

In this paper, we summarized our experiences of a two-year project where we sought to develop automated refactorings, and we made interesting observations about the opinions of the developers who utilized our tools. The results showed that they found most of the manual refactorings of coding issues easily implementable via automatic transformations. Also, when we implemented these transformations and observed the automated solutions, we found that almost all refactoring types helped them to improve their code.

We had to take into account several expectations of the developers when we designed and implemented the automatic refactoring tools. Among several challenges of the implementation, we identified some quite important ones, such as performance, indentation, formatting, understandability, precise problem detection, and the necessity of a precise syntax tree. Some of these have strong influence on the usability of a refactoring tool, hence they should be considered early on the design phase. Here, our recommendations may serve as a guideline for others to design and develop automatic refactoring tools that meet the high expectations of today's developers.

References

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

[2] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, "A Quantitative Evaluation of Maintainability Enhancement by Refactoring," in *Proc. of ICSME 2002*. IEEE, 2002, pp. 576–585.

[3] T. Mens and T. Tourwé, "A Survey of Software Refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[4] J. Ratzinger, M. Fischer, and H. Gall, "Improving Evolvability Through Refactoring," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[5] S. Demeyer, "Refactor Conditionals into Polymorphism: What's the Performance Cost of Introducing Virtual Calls?" in *Proc. of ICSM*. IEEE, 2005, pp. 627–630.

[6] K. Stroggylos and D. Spinellis, "Refactoring–Does It Improve Software Quality?" in *Proc. of the 5th Int. Workshop on Software Quality*. IEEE Comp. Soc., 2007, p. 10.

[7] M. Alshayeb, "Empirical Investigation of Refactoring Effect on Software Quality," *Inf. Softw. Technol.*, vol. 51, no. 9, pp. 1319–1326, Sep. 2009.

[8] A. Yamashita and L. Moonen, "To What Extent Can Maintenance Problems Be Predicted by Code Smell Detection? - An Empirical Study," *Inf. Softw. Technol.*, vol. 55, no. 12, pp. 2223–2242, Dec. 2013.

[9] M. Kim, T. Zimmermann, and N. Nagappan, "A Field Study of Refactoring Challenges and Benefits," in *Proc. of the 20th Int. Symposium on the Foundations of Software Engineering*. ACM, 2012, pp. 50:1–50:11.

[10] G. H. Pinto and F. Kamei, "What Programmers Say About Refactoring Tools?: An Empirical Investigation of Stack Overflow," in *Proc. of the 6th Workshop on Refactoring Tools*. ACM, 2013, pp. 33–36.

[11] D. Campbell and M. Miller, "Designing Refactoring Tools for Developers," in *Proc. of the 2nd Ws. on Refactoring Tools*. ACM, 2008.

[12] G. Szőke, G. Antal, C. Nagy, R. Ferenc, and T. Gyimóthy, "Bulk Fixing Coding Issues and Its Effects on Software Quality: Is It Worth Refactoring?" in *Proc. of SCAM 2014*. IEEE, 2014, pp. 95–104.

[13] G. Szőke, C. Nagy, R. Ferenc, and T. Gyimóthy, "A Case Study of Refactoring Large-Scale Industrial Systems to Efficiently Improve Source Code Quality," in *Proc. of ICCSA 2014*. Springer, 2014, pp. 524–540.

[14] G. Szőke, C. Nagy, P. Hegedűs, R. Ferenc, and T. Gyimóthy, "Do Automatic Refactorings Improve Maintainability? An Industrial Case Study," in *Proc. of ICSME 2015*. IEEE, 2015, pp. 429–438.

[15] G. Szőke, C. Nagy, L. J. Fülöp, R. Ferenc, and T. Gyimóthy, "Fault-Buster: An Automatic Code Smell Refactoring Toolset," in *Proc. of SCAM 2015*. IEEE, 2015, pp. 253–258.

[16] C. K. Roy, M. F. Zibran, and R. Koschke, "The Vision of Software Clone Management: Past, Present, and Future (keynote paper)," in *Proc. of CSMR-WCRE*. IEEE, 2014, pp. 18–33.