Towards an Appropriate Software Refactoring Tool Support

MARIJA KATIĆ, KREŠIMIR FERTALJ
Department of Applied Computing
Faculty of Electrical Engineering and Computing, University of Zagreb
Unska 3, 10 000 Zagreb
CROATIA
marija.katic@fer.hr, kresimir.fertalj@fer.hr http://www.zpr.fer.hr

Abstract: Refactoring is a technique that is concerned with changing internal software design while external software behaviour is preserved. Although the refactoring process seems to be clear and well defined, there are difficulties with regard to its automation. Various refactoring tools have been developed so far, which makes not easy for a developer to choose an appropriate tool to work with. This paper tries to compare some refactoring tools with respect to the automation and coverage, reliability, configurability, scalability and discoverability and so gives directions for the appropriate software refactoring tool selection.

Key-Words: refactoring, refactoring process automation, refactoring tool, refactoring tools evaluation, refactoring tools overview

1 Introduction
Software design is usually deteriorated due to added features. According to Lehman's laws of software evolution as a program is evolved its complexity increases and the quality decreases [11]. Refactoring has evolved to cope with this issue. Refactoring is an object-oriented variant of restructuring [18] [19], and restructuring is the transformation from one representation form to another at the same abstraction level, while preserving the system's external behaviour (functionality and semantics) [20]. Fowler has given a list of 72 refactorings (refactoring techniques) for the source code design improvement [10] and Kerievsky has introduced a list of refactoring to patterns techniques [21]. Although all software artefacts (documentation, design models, source code ...) can be exposed to the refactoring [17], we only consider the source code or program refactoring. The process of program refactoring consists of the three main steps: identification of source code parts that need to be restructured – "bad smells" [10], determination of the appropriate refactoring technique, and execution of the most appropriate technique that has been determined. Even though the refactoring process seems to be clear and well defined, there are difficulties with regard to its automation [27].

This paper is rather survey on refactoring tools and automation of the refactoring process than survey on the refactoring itself. It tries to evaluate refactoring tools, foremost, with regard to their usefulness to developers. Moreover, it gives evaluation criteria for refactoring tools that are specific and important for programmers as tool users. Extensive overview of software refactoring is given in the work of Mens and Tourwe [18] and also more general in the work of Katić and Fertalj [17].

The contribution of our work lie in evaluation criteria that we have chosen and conclusions that we have derived based upon the evaluation and comparison of selected tools. The main purpose is to indicate properties that a good software refactoring tool must have.

Chapter 2 gives a brief description of the refactoring process automation and refactoring tools, Chapter 3 points out to the problem of low usage of tools. Selection of tools, their evaluation, and the criteria description are given in the chapter 4. Related work and paper conclusion are placed in the last chapters.

2 Automating Refactoring Process
The whole refactoring process can be done manually, which can be tedious and long-lasting. Automating this process can reduce the significant amount of a manual work and increase the efficiency, reliability and accuracy of the whole process. Although the notion of refactoring had been known much earlier, the term refactoring was first introduced in the work of W.F. Opdyke [19]. He considered that refactoring cannot be completely automated because it cannot decide which refactorings to apply. Mens and Tourwe indicated that the degree of automation of a refactoring tool varies depending on which of the refactoring activities are supported by the tool, as well as the extent to which support for each of these activities is automated [18].

2.1 Refactoring Tools
Refactoring tool is one that changes a bad software design to a "good software design"[12]. There are plenty of refactoring tools supporting only a part of the
refactoring process. They suffer from issues associated with the level of automation, stages of the refactoring process supported or automated, the subset of refactorings that can be applied, and the complexity of the supported refactorings [4]. With respect to such a consideration, refactoring tools are semi-automated or fully-automated. With semi-automated tools the task of the developer is to identify which part of the software needs to be refactored, and to select the most appropriate refactoring to apply, while the actual application of the refactoring is automated [18]. Fully-automated refactoring tool should support the whole aforementioned refactoring process. However, it can be doubtful whether the whole refactoring process can be automated completely without requiring any human intervention. We consider also fully-automated those tools that are able to give a user refactoring proposals and not select the most appropriate one.

The first refactoring tool, Refactoring Browser for Smalltalk, is an example of semi-automated tool [22]. Today, all integrated development environments (IDEs), such as IntelliJ IDEA [8] or Eclipse [23], come with built-in refactoring browsers and also enable extension to various add-ins that have code analyzing capabilities so enabling bad smell detection and also refactoring proposals and application. Therefore, they can be considered as fully-automated tools. Tools that are only able to detect bad smells in the code, can also be considered semi-automated. Dudziak and Włoka found that for smelling the diversity of the tools was far broader than for restructuring [15]. Over the time the situation has changed. There are more and more refactoring tools actually providing analysis and transformation capabilities altogether.

The notion of generic transformation was inevitable and was firstly introduced in [26]. However, generic program refactoring meaning model that is applicable to different programming languages, or, we dare to say, all programming languages, was introduced by Läämel as a language parametric framework that can be instantiated for various, rather different languages such as Java, Prolog, Haskell and XML [25]. More recently, this idea has been extended to an approach that allows addition of new refactorings or modification of existing ones. In [1] a framework for an adaptive refactoring tool has been introduced. It allows for adaptability to refactor code written in various languages.

### 2.2 Research Approach to Refactoring Tools

As opposed to the tools presented in the Table 1, in literature there are more advanced approaches to the refactoring process automation.

Fully-automated tool Guru is a reverse engineering tool which is limited only to a restructuring of inheritance hierarchy into an optimal one for the objects currently in the system for Self programming language [24].

Semi-automated approach in [30] automatically finds places in the program that are candidates for specific refactorings and uses the Daikon tool for dynamically discovering program invariants to accomplish it.

Fully-automated approach in [31] goes beyond the Refactoring Browser [22] and integrates with it. The tool is able to a limited number of detect bad smells and propose adequate refactorings bases on detected smells. It supports a couple of composite refactorings such as one that is able to add class and push up method.

Jeon, Lee and Bae proposed a fully-automated approach based on design patterns in Java programs [13]. For a particular design pattern, they defined an inference rule to automatically identify a set of candidate spots (class or a set of classes) and a refactoring strategy to transform one of the set of candidate spots into the desired design pattern structure.

J/Art is a tool for detecting a number of structural weaknesses that was created by Dudziak and Włoka [15]. The tool is able, for certain problems, to determine the most appropriate refactorings and perform them automatically.

Clustering approach is a model that has been used by Czibula and Serban to recondition the class structure of a system [6]. They indicated that their technique is deterministic and consumes far less time than existing approaches.

### 3 Low Distribution of Refactoring Tools

It is a fact that refactoring tools are rarely used. This fact has also been considered by Murphy-Hill, Parnin and Black who describe it as a problem, because manual refactoring is slow and error-prone [5]. They state that the reason for that is poor usability of tool's user interface and focus is on efficiency, errors and satisfaction components of usability. Also they state that almost 90% of refactorings are performed manually without the help of tools. On the other hand, Mens and Tourwe use the word usability and characterize it with the automation, reliability, configurability, coverage and scalability characteristics [18]. Campbell and Miller, in their efforts of eliminating or reducing barriers between programmers and refactoring tools, have identified three of the most common barriers: discoverability, lack of trust or lack of familiarity and productivity [3].

Straightforward issue is to identify what is the most suitable refactoring tool for a programmer to use, meaning what is the tool that goes beyond above mentioned barriers. We have tried to answer on this question pointing out to requirements (evaluation criteria in section 4.1) that a good refactoring tool should have.
4 Evaluating Refactoring Tools Support

When searching for refactoring tools, we have encountered their expansion recently. This fact makes a selection of an appropriate refactoring tool even more difficult. We propose essential criteria that must be considered when choosing an appropriate tool to work with. While section 2.2 briefly describes some research approaches to the refactoring process automation, this section points out to refactoring tools commonly used by developers. The fact is that researchers and tool vendors rarely work together. There is a huge need to join these two streams, going in the same direction, and determine to which extent theoretical models can be applied on the real world projects. However, this topic goes beyond our scope. We have decided to choose those tools that are add-ins to integrated development environments and searched for a couple of them that are popular and used by developer's community. Roberts, Brant and Johnson specified the need of refactorings to be integrated into the standard development tools [22]. Tools that are considered in this work are as follows: Refactoring Browser [22], Resharper [9], Refactor [28], nDepend [14], Refactorit [2], IntelliJ Idea [8] and Eclipse [23]. Resharper, Refactor, nDepend and IntelliJ Idea are commercial tools, Refactorit and Eclipse are an open source projects and Refactoring Browser is a developed research tool.

4.1 Evaluation Criteria

In accordance with the research and testing of selected tools that we performed, we have chosen the criteria as follows:

- Automation and coverage – indicates the degree of the automation (fully-automated or semi-automated tool) as well as the activity of the refactoring process that is automated (bad smell identification, refactoring selection/proposal and refactoring application);
- Reliability – points out to the program behaviour preservation for supported refactorings (extract class, move method, replace temp with query, …) – we have checked out tool support for behaviour tests and code preservation which are undo and redo operations; also we have checked a support for a history of performed refactorings;
- Configurability – user-defined specifications on refactorings, meaning is there a possibility for user to add additional refactorings and is it possible to configure bad smell and refactoring connection;
- Scalability – refers to the elementary program transformations or primitive refactorings (extract methods, rename method, …), when they are applied sequentially, they are called composite refactorings (explained below) and as such they are able to increase scalability and performance [18]; we have checked support for primitive or composite refactorings and tool possibility to specify sequential dependencies between refactorings when there are performed sequentially, as well as are computations for bad smells discovery intensive or negligible;
- Discoverability – in a background, code analysis is usually performed and based upon that proposals for refactorings are given; we have checked support for a guidance in identifying how refactoring might be useful (is there a description of refactoring and contribution of its application, or is there a description of a bad smell), also is there a highlighting mechanism of code smells; we have also checked a preview mechanism to show changes before their application.

Replace conditional with polymorphism is an example of the composite refactoring and consists of several primitive refactorings (extract method, move method, replace type code with subclasses) that need to be applied sequentially in order to accomplish the polymorphism concept.

There are various formalisms and techniques on how to describe refactorings and find bad smells in code, such as graph transformation [29], clustering [6], computation of invariants [30] in dynamic analysis, distance based cohesion metric [31] in metrics based refactoring tools. We have decided not to include them to the set of evaluation criteria, because in literature there is no consensus on which technique is better than others.

According to the evaluation criteria we have made a comparable table that contains all relevant information for the selected tools, Table 1.

4.2 Lessons Learned

The conclusions that we are able to draw with respect to our research, are as follows:

- Most refactoring tools are provided with only a limited set of automated activities, usually only with a refactoring application. However, mostly a limited scope of bad smells identification is automated and a user intervention is inevitable, especially when choosing which refactoring to apply.
- Predefined set of refactorings (e.g. extract method, rename method, push down variable into subclass, rename temp with query …) is automated and contemporary solutions can be extended, again, to a limited set of techniques. The future of the refactoring process automation is generic and adaptive approach to refactoring, such as proposed in [1].
• Tools integrate with most common testing environments and they are able to automatically search for unit and integration tests, but still not able to generate test cases based upon performed transformations. Furthermore, undo and redo operations guarantee code preservation and they are supported by most tools as well as history of performed refactorings.

• It seems that composite refactorings can be constructed easily because most tools include an API which can be used to add new refactorings or to call existing ones. However, there are difficulties such as behaviour preservation, preconditions and post conditions that must be satisfied and that make this process hard to implement. In spite of proposed solutions in the literature [31], there is a lack of such concepts used in everyday programmer's work.

• Refactoring tools are immature regarding configuration of bad smells and refactorings, moreover because they are limited in bed smells detection.

• There are more tools that support primitive refactorings than those that support composite refactorings. Consequently, they are not able to detect the sequence of possible refactorings. On the other hand, some of them are provided with a metric that can prove the performance gain.

• In order to take a full advantage of tools, we have found that it is the best solution to combine them. For example, for a developer working in the Visual Studio .NET it is great idea to combine nDepend [14] and Resharper [9] tools.

5 Related Work

There is no defined comparison framework for refactoring, but frameworks for assessing software evolution or general development environment can be used [4]. The taxonomy of software evolution is an example of assessing software evolution [16]. It includes categorization consisting of the four logical groups: when do we make a change, where, what we change, and change support. The use of taxonomy was demonstrated by applying it to three different software evolution tools: the Refactoring Browser, the Concurrent Version System, and eLiza self-managing servers [16]. Afterwards, the taxonomy was applied to the four refactoring tools: Smaltalk, VisualWorks 7.0, Eclipse 2.0, Guru and Together ControlCenter 6.0 [7]. The differences that were found were minor.

Glynn and Strooper go a step forward in assessing software refactoring tools and present a framework based on the DESMET method [4]. They tend to overcome issues with existing evaluation frameworks, such as the lack of specific criteria for software refactoring, the lack of formal process for application or their incompleteness with regard to user interaction and usability requirements. They have developed nine evaluation criteria: supplier assessment, maturity of tool, economic issues, ease of introduction, reliability, efficiency, usability, compatibility, and refactoring specific. Eclipse, IntelliJ Idea, Borland Jbuilder, RefactorIt, Condenser, and JCosmo were evaluated. The conclusions were that existing support for refactoring is inadequate due its incompleteness, and in the case of code smell detection tools, their immaturity.

Existing evaluation frameworks are too broad for refactoring specific tools, if we exclude the approach from [7] which is more specific and considers the levels of automation for the different stages of refactoring process and customisability allowing definition of additional code smells or refactorings, but again broad considering all supported artefacts. Our approach considers only tool support for the source code refactoring and evaluates it according to criteria that are more close to the developer who uses the tool.

6 Conclusion

This paper gives a brief overview of current extensive refactoring tools support and comparison of a number of tools according to the criteria specified. It can be difficult for a developer to choose an appropriate tool between lots of them. We have tried to point out to some criteria that should be considered in such decisions. Very often programmers will not use a tool because they are not sure how it will transform their code. It is hard to say which tool is the best, because there is no tool that conforms to all requirements in the best way. It is better to say that it depends on the developer's preference to specific criteria. Therefore we have pointed out criteria we think that are specific and important for programmers as refactoring tool users.

However, we can conclude that refactoring tool support is still immature, even though there are plenty of tools developed. Mature refactoring tool first of all should support all three steps of the refactoring process i.e. be fully-automated, should preserve program behaviour and provide support for program behaviour tests including automated tests generation, should enable undo and redo operations, should be configurable not only regarding refactoring extensions, but also regarding code smells extensions and their connections and definitely should support composite refactorings and their addition. In meantime, current tools can be combined in order to take a full advantage of all automated features.
<table>
<thead>
<tr>
<th>Refactoring Browser (Smalltalk)</th>
<th>Automation and coverage</th>
<th>Reliability</th>
<th>Configurability</th>
<th>Scalability</th>
<th>Discoverability</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semi-automated</td>
<td>Support for program behaviour tests?</td>
<td>No, behaviour is preserved only when a set of preconditions is provided</td>
<td>No, but programming extensions are possible such as in [31]</td>
<td>Primitive refactorings, [31] supports composition</td>
<td>No</td>
<td>[22]</td>
</tr>
<tr>
<td>Refactoring application and search for duplicated or unused code</td>
<td>Code preservation (undo/redo)?</td>
<td>No</td>
<td>No, Smalltalk provides a continuous change history</td>
<td>Yes, with respect to the manual refactoring and computation is negligible</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Refactoring history?</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Re handleMessage (C#)</th>
<th>Automation and coverage</th>
<th>Reliability</th>
<th>Configurability</th>
<th>Scalability</th>
<th>Discoverability</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fully-automated</td>
<td>Support for program behaviour tests?</td>
<td>Yes (automated search for all unit tests, integration for NUnit, MSTest, …)</td>
<td>No, but open API enable addition of new refactorings</td>
<td>Primitive refactorings</td>
<td>Poor</td>
<td>[9]</td>
</tr>
<tr>
<td>Refactoring, limited bad smells identification and refactoring proposals</td>
<td>Code preservation (undo/redo)?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Poor and limited smells (method is never used, qualifier is redundant, …)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Refactoring history?</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resharper (C#, VB.NET, C++, Java)</th>
<th>Automation and coverage</th>
<th>Reliability</th>
<th>Configurability</th>
<th>Scalability</th>
<th>Discoverability</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fully-automated</td>
<td>Support for program behaviour tests?</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes (powerful)</td>
<td>[3]</td>
</tr>
<tr>
<td>Refactoring, limited bad smells identification and refactoring proposals</td>
<td>Code preservation (undo/redo)?</td>
<td>Yes, but provided with IDE and on all changed files</td>
<td>Yes</td>
<td>No</td>
<td>Poor and limited smells (type can be moved to separated file, …)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Refactoring history?</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>사업소나포 (C#)</th>
<th>Automation and coverage</th>
<th>Reliability</th>
<th>Configurability</th>
<th>Scalability</th>
<th>Discoverability</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semi-automated</td>
<td>Support for program behaviour tests?</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes, for each defined metric provides an appropriate recommendation (e.g. if method code lines is higher than 20)</td>
<td>[14]</td>
</tr>
<tr>
<td>Primarily code-metrics tool</td>
<td>Code preservation (undo/redo)?</td>
<td>No</td>
<td>Yes, big detections can be used for bad smells identification by querying the code</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Bad smells identification (automatically generates information that give advices for refactoring)</td>
<td>Refactoring history?</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Refactor (C++, Java)</th>
<th>Automation and coverage</th>
<th>Reliability</th>
<th>Configurability</th>
<th>Scalability</th>
<th>Discoverability</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semi-automated</td>
<td>Support for program behaviour tests?</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>[2]</td>
</tr>
<tr>
<td>Bad smells directions (based on metric results a user can decide which refactoring will perform)</td>
<td>Code preservation (undo/redo)?</td>
<td>No as stand-alone, but as add-in, for example, JUnit can be combined with Yes, on all changed files</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Refactoring history?</td>
<td>No</td>
<td>Yes, as it is open source it is possible to contribute</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Eclipse (C++, Java)</th>
<th>Automation and coverage</th>
<th>Reliability</th>
<th>Configurability</th>
<th>Scalability</th>
<th>Discoverability</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fully-automated</td>
<td>Support for program behaviour tests?</td>
<td>Yes (integration for JUnit, …)</td>
<td>No, but provides an API for new refactorings construction</td>
<td>Primitive refactorings</td>
<td>Poor</td>
<td>[23]</td>
</tr>
<tr>
<td>Refactoring, limited bad smells identification and refactoring proposals</td>
<td>Code preservation (undo/redo)?</td>
<td>Yes, on all changed files, and only in the case when files are not modified and saved</td>
<td>No</td>
<td>Yes, with respect to the manual refactoring and computation is negligible</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Refactoring history?</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IntelliJ IDEA (Java)</th>
<th>Automation and coverage</th>
<th>Reliability</th>
<th>Configurability</th>
<th>Scalability</th>
<th>Discoverability</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fully-automated</td>
<td>Support for program behaviour tests?</td>
<td>Yes (automated search for all unit tests, integration for JUnit, TestNG, …)</td>
<td>No, but provides interfaces for invoking IDEA refactorings</td>
<td>Primitive refactorings</td>
<td>Poor</td>
<td>[8]</td>
</tr>
<tr>
<td>Refactoring, limited bad smells identification and refactoring proposals</td>
<td>Code preservation (undo/redo)?</td>
<td>Yes, on all changed files</td>
<td>No</td>
<td>Yes, with respect to the manual refactoring and computation is negligible</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Refactoring history?</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

Table 1
References:


[19] W. Opdyke, Refactoring Object-Oriented Frameworks, PhD Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1992


