Volume 6, Issue 1, January 2016 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Fascinating Perspective of Code Refactoring

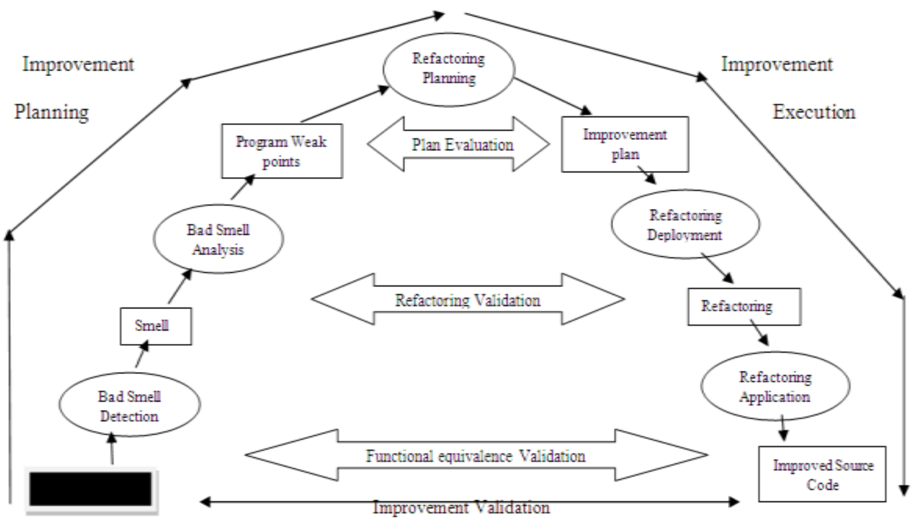
Dr. V. Sangeetha (M.Sc., Mphil)[1], **M. Sangeetha (M.Sc., Mphil)**[2]
[1] Asst. Professor, Department of Computer Science, Prucas, Dharmapuri, Tamilnadu, India
[2] Head cum Asst. Professor, Department of Computer Science Shakthikailash, Dharmapuri, Tamilnadu, India

***Abstract:** There is a constant need for practical, efficient, and cost effective software evaluation techniques. As the application code becomes older & older, maintaining it becomes a challenge for the enterprises due to increased cost of any further change in it. Refactoring is a technique to keep the code cleaner, simpler, extendable, reusable and maintainable. Refactoring includes code refactoring to achieve removal of unused code and classes, renaming of classes methods and variables which are misleading or confusing. Code Standard Refactoring includes code refactoring to achieve the quality code. Developers should regularly refactor the code as per the Standard code lines. Refactoring leads to constant improvement in software quality while providing reusable, modular and service oriented components. It is a disciplined and controlled technique for improving the software code by changing the internal structure of code without affecting the functionalities. Code is not easily maintainable, extending/adding new features in the application are not possible or very expensive, Application is using older version of software's instead of using latest version and hence new features can't be used and explored in the application. It also transforms a program to improve its internal structure, design, simplicity understandability or other features without affecting its external behavior. Implementing a refactoring tool is a real challenge and even today's the most mature tool implementations are far from being bug-free. Researchers have made various proposals within recent years how these challenges can be approached. But unfortunately there is a lack of communication between researches and tool developers. This paper reveals the research pattern, common concerns, and statistics to formulate better research topics.*

***Keywords:** software refactoring, code smell detection, automated tools, eclipse*

I. INTRODUCTION

"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." Refactoring is typically done in small steps. After each small step, we're left with a working system that's functionally unchanged. Practitioners typically interleave bug fixes and feature additions between these steps. So refactoring doesn't preclude changing functionality, it just says that it's a different activity from rearranging code. The key insight is that it's easier to rearrange the code correctly if we don't simultaneously try to change its functionality. The secondary insight is that it's easier to change functionality when we have clean (refactored) code. Refactoring is a kind of reorganization. Technically, it comes from mathematics when we factor an expression into equivalence; the factors are cleaner ways of expressing the same statement. Refactoring implies equivalence; the beginning and end products must be functionally identical. Practically, refactoring means making code clearer, cleaner, simpler and elegant or, in other words, clean up after our self when we code. Examples would run the range from renaming a variable to introducing a method into a third-party class that we don't have source for (W. G. Griswold *et al*., 1991). Refactoring (Tom Mens *et al*. 2004) is not rewriting, although many people think they are the same.



There are many good reasons to distinguish them, such as regression test requirements and knowledge of system functionality. The technical difference between the two is that refactoring, as stated above, doesn't change the functionality (or information content) of the system whereas rewriting does. Rewriting is reworking. Refactoring is a good thing because complex expressions are typically built from simpler, more gradable components. Refactoring either exposes those simpler components or reduces them to the more efficient complex expression (depending on which way we are going). The major part of the total software development cost is bestowed to software maintenance. Best software methods or tools do not solve this problem. Increasing new requirements the software becomes more complex. Managing complexity is important technical topic in software development. The research domain that addresses this problem is referred to restructuring or refactoring.

Refactoring, the process of changing the structure of code without changing the way a program behaves, is a potentially useful technique for building and maintaining code. For example, using the Extract Method refactoring, a programmer may remove duplicated code by putting it into a new method and calling that new method instead. Semi-automated refactoring tools in most commercial programming environments relieve the programmer from having to make tedious and error-prone changes by hand. For example, a tool can encapsulate a field using getter and setter methods, and the tool will automatically replace all direct references to the field with references to the new methods. In this way, a refactoring tool offers the potential for tremendous increases in refactoring productivity.

Refactoring can be semi-automated with the help of tools, but many existing tools do a poor job of communicating errors triggered by the programmer. This poor communication causes programmers to refactor slowly, conservatively, and incorrectly. In this paper, we demonstrate the problems with current refactoring tools, characterize three new tools to assist in the Extract Method refactoring, and describe a user study that compares these new tools to existing tools. The results of the study show that these new tools increase both the speed and accuracy of refactoring. Based on the new tools and my observation of programmers, I present several guidelines to help build future refactoring tools.

II. WHY WE NEED REFACTORING

Refactoring improves code quality, reliability and maintainability throughout software life cycle.

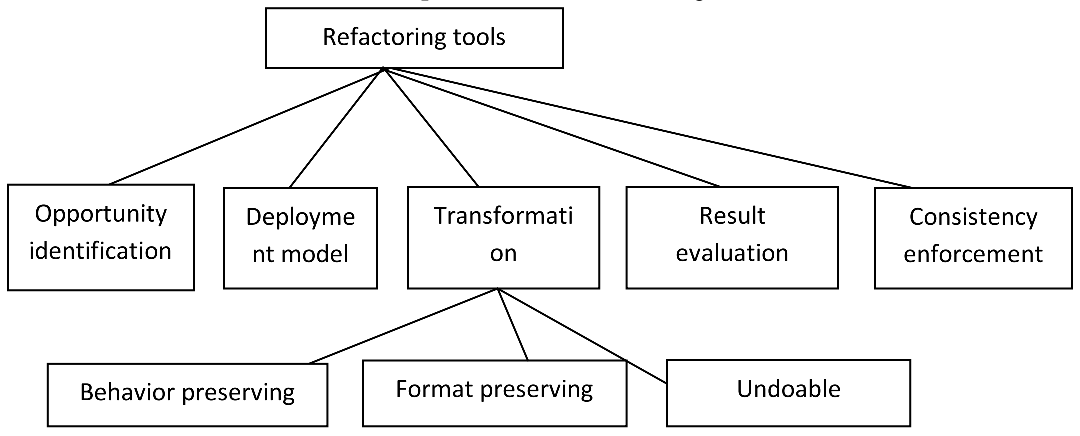
1. Refactoring improves the design of software
2. Refactoring makes software easier to understand
3. Refactoring helps find the bugs
4. Refactoring helps to program faster

III. REFACTORING ACTIVITIES

The refactoring process consists of number of different activities, each of which can be automated to certain extent:

1. Identify where the software should be refactored
2. Determined which refactoring should be applied to the identified places
3. Guarantee that the applied refactoring perverse behavior
4. Apply the refactoring

Functional requirements of refactoring tools



a. *Identification of Refactoring Opportunities:*

Opportunities are potential design improvements. They can be detected manually or via code smells, design and anti-patterns or generally just via any kind of applicable metric.

b. *Deployment Model:*

A refactoring tool should be easy to use and always at hand. Refactoring available directly within an IDE mitigate the need to switch to another tool, which in return increases developer productivity.

c. *Transformation:*

The tool has to transform the code according to the selected refactoring and should fulfill the following qualities:

- **Behavior Preserving:** Refactoring tools need to assert the correctness of applied transformations. If programmers do not quickly gain a certain degree of trust in a tool, they do not use it. Behavior preservation furthermore reduces the need to execute automated tests after each small code modification. This can speed up the refactoring process significantly.

- **Format Preserving:** Refactoring should not change code arbitrary. A tool must retain the formatting of all not directly affected statements. Comments must be preserved.
- **Undoable:** Refactoring lend to be used in an exploratory manner. A user should not just be able to apply a refactoring easily, but should also be able to undo it, if the result is not as expected.

d. **Result Evaluation*:*** A tool must provide means to the user to understand what code was changed and why.

e. ***Consistency Enforcement:*** Refactoring change code and should ideally also maintain horizontal and vertical consistency.

IV. TOOLS SUPPORT FOR REFACTORING

Tools support is crucial for the success of bad smell detection and resolution. Fully-automatic tools complete the multi-stage refactoring process without user interaction, from initial identification of where it is needed through the selection and application of a specific refactoring. Static, fully automated software refactoring tools focus on detection and removal of duplicated code. In the Guru fully automatic refactoring tool, restructuring of inheritance hierarchies is also performed automatically for programs written in the Self programming language (the Java Equivalent is Condenser. Although fully-automated tools assist software developers, the lack of user input into the process causes the introduction of meaningless identifier names, lack of customizability, and negative impacts on a user's current understanding of a system Semi automated refactoring tools attempt to address the problems raised by fully-automated or fully-manual processes by retaining user input to guide the refactoring process whilst automating the tedious, error-prone and complex sub-tasks. Current support for semi-automated software refactoring focuses on a single stage of the refactoring process, e.g., the implementation of refactoring such as 'Extract method'. This single-stage focus unnecessarily burdens developers with tasks that could be completed automatically, Such as identifying where a refactoring may be applied. Additionally, tools automating different stages of refactoring are not themselves integrated, making the linking Together of the stages (i.e., passing the output of one stage into the next) the task of the user). Examples of semi automated refactoring are the refactoring transformation support in environments such as Eclipse, and the code smell detection of jCOSMO .

V. DETECTION OF BAD SMELL

Tool support is crucial for the success of bad smell detection and resolution because of the following reasons. First, uncovering bad smells in large systems necessitates the use of detection tools because manually uncovering these smells is tedious and time-consuming, especially those involving more than one file or package, e.g., duplicated code. The tools are expected to detect bad smells automatically or semi automatically. Clone detection is an excellent example, and researchers have proposed detection algorithms and developed tools for clone detection in the last decades. Second, software engineers need tools to automatically or semi automatically carry out refactoring to clean bad consuming and error prone. For example, renaming a variable requires revising all references to that variable. Manually identifying all references is challenging—an issue that detection tools based on program analysis seeks to address. Most mainstream integrated development environments (IDE), such as Eclipse Microsoft Visual Studio and IntelliJ IDEA support software refactoring. Professional refactoring tools have also been developed. Rich tool support, in turn, accelerates the popularization of software refactoring. Human intervention, however, remains indispensable to bad smell detection and resolution because of the following reasons.

First, most bad smells automatically detected should be rechecked manually because 100 percent precision cannot be guaranteed by detection tools. Second, it is up to software engineers to determine how to restructure bad smells in terms of refactoring rules that should be applied, and arguments of the rules.

VI. WHY USE AN AUTOMATED TOOL?

When doing refactoring the externally observable behavior must be guaranteed to stay the same. If refactoring is carried out manually, one needs to frequently rebuild the system and run tests. Manual refactoring is therefore really practical only, when the following conditions hold:

1. The system, of which the refactored code is a part, can be rebuilt *quickly.*
2. There are automated "regression" tests that can be frequently run.

This situation is not very common, meaning that the applications of refactoring are limited. This situation is becoming more common, particularly as more people use XP (Extreme Programming) development methods. Another hindrance is that many of these refactoring are tedious. Potentially requiring hours of precious development time, as the change is made and then thoroughly checked. Not many programmers would enjoy the task of renaming a method in a large code base. A simple search and replace will potentially find extra results. So each replacement must be examined by the programmer. However there is no great intelligence to the operation all that is wanted is to rename any use of a method on a given class or its subclasses. A refactoring tool therefore can save hours of work. Even more importantly give confidence that the correct changes were made.

The speed of automated tools has another benefit, which is shown in team development environments. A developer is much less likely to perform refactoring operations if the source code involved is under the responsibility of other developers as well. However by using an automated tool, the refactoring can be completed quickly so that other developers are not held up waiting to make their changes when the refactoring is completed, or worse making their changes on the old code at the same time as the refactoring operation. This ensures that even when the responsibility for a section of code is shared, developers will not reach a stale mate, where none of the developers make the required changes. Integration with the developers chosen IDE also bring many benefits. First having the tools at hand, means that developers can more easily refactor. They do not have to switch between development and refactoring modes, and can instead see it as part of their normal development cycle. Secondly IDE features such as Source Control Integration can reduce the effort in refactoring such as move class or rename package.

VII. LIST OF AUTOMATED CODE REFACTORING TOOLS

Many software editors and IDEs have automated refactoring support. Here is a list of a few of these editors, or so-called refactoring browsers.

- IntelliJ IDEA (for Java)
- WebStorm (for JavaScript)
- Eclipse
- NetBeans (for Java)
- JDeveloper (for Java)
- Embarcadero Delphi
- Visual Studio (for .NET and C++)
- Photran (a Fortran plugin for the Eclipse IDE)

VIII. REFACTORING IN ECLIPSE

Eclipse was among the first IDEs to help bring refactoring to the mainstream developer. Eclipse version 1.0 included several highly useful Java refactoring, which are nowadays staple tools in most Java developers' toolbox. These included Extract Method, Rename and Move. Eclipse 2.0 added statement-level refactoring such as Extract and Inline Local Variable and also higher-level ones, e.g., Change Method Signature, and Encapsulate Field. Some refactoring, such as Rename, offer great leverage because of the potential scale of the changes they perform automatically. Others, like Extract Method, are more local in scope, but relieve the developer from performing the analysis required to ensure that program behavior is unaffected. In both cases, the developer benefits from reduction of a complex and numerous changes to a single operation.This help to maintain his focus on the big picture. Moreover, the ability to quickly roll back the changes enables exploration of design possibilities more easily, and without fear of irreparable damage to the code base. Eclipse 2.1 included several type-oriented refactoring such as Extract Interface and Generalize Type that address the problems of both scale and analytic complexity. These used a common analysis framework [8] based on theoretical work from Palsberget al. [6] for expressing the system of constraints that ensure the type-correctness of the resulting program. Such frameworks are important because they speed the development of entire families of refactoring, e.g., [7]. Our belief is that the incorporation of reusable and extensible frameworks for the various classical static analyses (type, pointer, data flow) into Eclipse will be critical to the expansion of the suite of refactoring. Eclipse 3.0 introduced refactoring over multiple artifact types, which in part dealt with a problem faced by Eclipse plug-in developers: the Rename refactoring had been previously oblivious to references located in plug-in metadata, so that renaming an extension implementation class would break the reference, leaving the extension class unreachable by the extension point framework. Since extension points are the sole mechanism for providing functionality in Eclipse, this was a serious problem. As a solution, the Eclipse Language Toolkit (LTK) provided a "participant" mechanism, allowing additional entities to register interest in a given type of refactoring, and participate in both checking pre-conditions and contributing to the set of changes required to effect the refactoring. Using this mechanism, breakpoints, launch configurations, and other artifacts outside the source itself can be kept in sync with source changes. As applications are increasingly built using multiple languages, this ability becomes critical to the applicability of automated refactoring to the mainstream developer. Eclipse 3.1 included Infer Type Arguments [3], a migration refactoring that helps Java 5 developers migrate client code of libraries to which type parameters have been added. The migration is important because it increases static type safety. The necessary analysis, however, is subtle and pervasive enough that many developers might hesitate to perform the migration manually. Of particular interest was the Java 5 Collections library, which had been retrofitted with type parameters. In particular, the Infer Type Arguments refactoring infers the types of objects that actually flow into and out of the instances of these parametric types, and inserts the appropriate type arguments into the corresponding variable declarations as needed. In some sense, the underlying analysis reconstructs an enhanced model of the original application, recovering lost or implicit information that may be critical to maintenance or further development. As such, this kind of refactoring offers benefits in maintaining or even "revitalizing" legacy code. Before Infer Type Arguments can be applied, however, the library itself must be parameterized. Java 5 Collections were parameterized manually, but many other existing libraries would benefit from added type-safety and expressiveness, if they were converted to use generics. A recently described refactoring, Introduce Type Parameter [5], addresses this complex issue. With the addition of such a refactoring, the Eclipse JDT would support developers in a wide spectrum of generics-related maintenance tasks. Eclipse 3.2 introduced a team-oriented innovation: storing API refactoring with the library itself, along with a "playback" mechanism to automatically perform the necessary transformations on API client code when the new library is imported [4, 1]. Such tools help smooth the interactions amongst team members and between teams, by automatically propagating changes from one component to another, or by automatically making the necessary changes implied by another. As software development becomes increasingly distributed, we believe this sort of tooling will become vital. Eclipse 3.3 offers an Introduce Parameter Object refactoring. Additionally, a great number of Cleanups that can also be applied to source files on save, for example Organize Imports, Format, or adding missing J2SE-5-style annotations.

IX. CONCLUSION

Refactoring is a well defined process that improves the quality of systems and allows developers to repair code that is becoming hard to maintain, without throwing away the existing source code and starting again. By careful application of refactoring the system's behavior will remain the same, but return to a well structured design. The use of automated refactoring tools makes it more likely that the developer will perform the necessary refactoring, since the tools are much quicker and reduce the chance of introducing bugs.

REFERENCE

[1] D. Dig. Using refactoring to automatically update component-based applications. In *OOPSLA Companion*, pages 228–230, 2005.

[2] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[3] R. Fuhrer, F. Tip, A. Kie˙zun, J. Dolby, and M. Keller. Efficiently refactoring Java applications to use generic libraries. In *ECOOP*, pages 71–96, July 2005.

[4] J. Henkel and A. Diwan. Catchup! capturing and replaying refactorings to support API evolution. In *ICSE*, pages 274–283, May 2005.

[5] A. Kie˙zun, M. D. Ernst, F. Tip, and R. M. Fuhrer. Refactoring for parameterizing Java classes. In *ICSE*, May 2007.

[6] J. Palsberg and M. I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley and Sons, 1994.

[7] F. Steimann, P. Mayer, and A. Meißner. Decoupling classes with inferred interfaces. In *SAC*, pages 1404– 1408, 2006.

[8] F. Tip, A. Kie˙zun, and D. B¨aumer. Refactoring for generalization using type constraints. In *OOPSLA*, pages 13–26, Nov. 2003.

ABOUT AUTHOR

M.SANGEETHA M.Sc.,M.Phil(CS).,
Head cum Assistant Professor
Department of Computer Science
Sakthikailash women's college, Dharmapuri
E-mail :mslion2010@gamil.com
Research area: Software Engineering