International Journal of Current Engineering and Technology
E-ISSN 2277 – 4106, P-ISSN 2347 – 5161

Available at http://inpressco.com/category/ijcet

Review Article

A Review on Code Cleanup and Code Standard Refactoring

Ganesh B. Regulwar†* and R. M. Tugnayat‡

†BNCOE, Pusad, Maharashtra, India
‡SSPACE, Wardha, Maharashtra, India

Accepted 31 July 2015, Available online 10 Aug 2015, **Vol.5, No.4 (Aug 2015)**

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. Code Clean Up 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.

Keywords: *Code clean up, code standard, maintainability, extendibility, improve software quality*

1. 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).

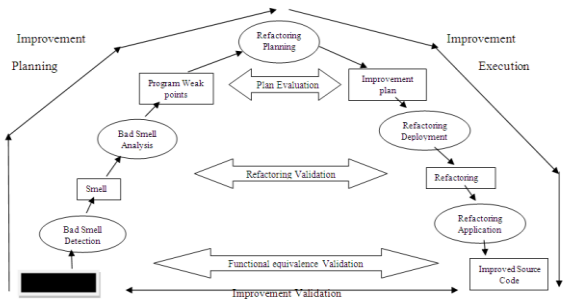

Figure1: The Refactoring Process

*Corresponding author **Ganesh B. Regulwar** is working as Assistant Professor and **Dr. R. M. Tugnayat** as **Principal**

For an example of efficiency, count the terms and operators: (x - 1) * (x + 1) = x^2 - 1. Four terms versus three & three operators versus two. However, the left hand side expression is (arguably) simpler to understand because it uses simpler operations. Also, it provides we more information about the structure of the function f(x) = x^2 - 1, like the roots are +/- 1, that would be difficult to determine just by "looking" at the right hand side.

Refactoring tools have two (Gabriele Bavota *et al*. 2010) principal use cases. The first is refactoring in the small, e.g. interactive application by a programmer while operating in an IDE ("refactoring browser"). The second is refactoring in the large, e.g. off-line applications of single or multiple sets of refactorings by batch tools. The first is a useful convenience. The second can substitute entirely for the first, and can additionally carry out code modifications simply not practical by manual or even interactive refactoring.
Software refactoring or rewriting becomes essential for the organization when following problems becomes visible in the software:

Maintainability- code is not easily maintainable

Extendibility- extending/adding new features in the application are not possible or very expensive.

Use of old version of third party application- 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

2. Literature of View

According to Opdyke (W. F. Opdyke 1992), each refactoring basically consists of preconditions, mechanics and postconditions. All preconditions must be satisfied before applying refactoring. Likewise, all postconditions must be met after refactoring is applied. These conditions ensure that the program behavior is preserved. Opdyke also categorizes refactorings into low-level and high-level refactorings. Low-level refactorings are related to changing a program entity (e.g., create, move, delete). High-level refactoring are usually sequences of low-level refactorings. He also provides proofs of behavior preservation for many refactorings. The behavior preservation proofs of some low-level refactorings are trivial but implementing them is not as trivial.

2.1 Refactoring Tools

In early 1990s, Don Roberts and his colleagues developed a refactoring tool called theSmalltalk Refactoring Browser (D. Roberts *et al*. 1997). This refactoring browser allows the user to perform many interesting refactorings automatically (e.g., Rename, Extract/Inline Method, Add/Remove Parameter). However, this early tool was not popular because it was a stand-alone tool separate from the integrated development environment (IDE). Developers found it inconvenient to switch back and forth between the IDE (develop code) and Refactoring Browser (refactor code). Thus later refactoring tools (G.C. Murphy *et al*.2006) have been integrated in the IDEs. The following are refactoring tools for Java (Katsuhisa Maruyama *et al*. 2011) .

IntelliJ IDEA (IntelliJ. IDEA , 2002)This is an expensive commercial IDE. This tool also supports Rename and Move Program Entities (e.g., package, class, method, field), Change Method Signature, Extract Method, Inline Method, Introduce Variable, Introduce Field, Inline Local Variable, Extract Interface, Extract Superclass, Encapsulate Fields, Pull Up Members, Push Down Members and Replace Inheritance with Delegation.

RefactorItRefactorIt is commercial software that supports many automatic refactorings (Aqris. RefactorIt , 2001). It can cooperate with Sun ONE Studio, Oracle 9i JDeveloper and Borland JBuilder. The supported refactorings are: Rename, Move Class, Move Method, Encapsulate Field, Create Factory Method, Extract Method, Extract Superclass/Interface, Minimize Access Rights, Clean Imports, Create Constructor, Pull Up/Push Down Members.

JRefactory (sourceforge.net.JRefactory, 2003) JRefactory is a tool that is first developed by Chris Seguin. However, Mike Atkinson has taken over the leadership role since late 2002. This tool allows easy application of refactorings by providing user interface based on UML diagrams as visualization of Java classes. It can cooperate with JBuilder and Elixir IDEs. JRefactory supports the following refactorings: Move Class, Rename Class, Add an Abstract Superclass, Remove Class, Push Up Field, Pull Down Field, Move Method.

jFactor (Instantiations.jFactor , 2002) jFactor for VisualAge Java is a commercial product that provide a set of refactorings. Extract Method, Rename Method Variables, Introduce Explaining Variable, Inline Temp, Inline Method, Rename Method, Pull Up/Push Down Method, Rename Field, Pull Up/Push Down Field, Encapsulate Field, Extract Superclass/Interface.

Transmogrify (sourceforge.net.Transmogrify, 2001) Transmogrify is a Java source analysis and manipulation tool. This tool is under development and currently is focused on the refactoring tool. It is available as a plug-in for JBuilder and Forte4Java. It supports a limited set of refactorings such as Extract Method, Replace Temp with Query, Inline Temp, Pull Up Field.

Eclipse (eclipse.org. Eclipse(2003) Eclipse is a generic development environment by IBM. It also has refactoring support and some analysis. This project is open source. The current version of Eclipse (Helio version 3.6.1) supports many types of refactorings which include Move Method, Rename Method, Encapsulate Fields and etc.

Microsoft Visual Studio Microsoft Visual Studio is an integrated development environment.It supports many

primitive refactorings. Generally, refactoring tools (E. Murphy-Hill *et al*. 2008) provide a list of refactorings in which the user can choose from the menu. Once the user chooses which refactoring (Frank Tip *et al*. 2011) to apply, the tool performs analyses in the background checking the required conditions. If those conditions are met, then refactoring can be performed. A few tools, like the refactoring support in Eclipse, allow the user to preview the resulting code before committing changes to the code. The preview feature gives the user a better idea of which part of his code will be affected by such a refactoring and whether it corresponds with his intention.

There are many refactoring tools for other programming languages like C++. For instance, Xrefactory also known as xref (XRef-Tech. XRefactory , 1998) is a refactoring browser for Emacs, XEmacs and jEdit. CppRefactory (sourceforge.net.CppRefactory , 2001) is another open source refactoring tool that automates the refactoring process in a C++ project. Though the refactoring framework was originally proposed for the object-oriented (K.J. Lieberherr *et al*. 1989, P. L. Bergstein *et al*.1997, S. Tichelaar *et al*. 2001) programming language, many researchers apply the idea of refactoring to other language paradigms as well. Li and his colleagues propose refactoring tool support for functional languages (Huiqing Li *et al*. 2003) . Saadeh and Kourie (EmmadSaadeh *et al*. , 2009) developed a refactoring tool for Prolog. The general idea is similar to that of object-oriented (K.J. Lieberherr *et al*. 1989, S. Tichelaar *et al*. 2001) languages but the conditions and mechanics are different. Although most tools discussed in this research apply refactorings by directly manipulating the source code, many software designers think about refactorings at the design level. Researchers who are interested in design-level transformations include Griswold and Bowdidge (William G. Griswold *et al*. 1993). They state that it is rather difficult to conceptualize program structure by just observing the program text. Instead of using program text, a graphical representation of program structure is used as it permits direct manipulation of the program structure at design level.(Gorp *et al*. 2003, Enckevort *et al*. 2009 and Saadeh *et al*. 2009) and propose techniques to apply refactorings to UML diagrams. We omit further discussion regarding design-level transformations, since our work uses source code manipulation approach.

Like (Opdyke and Fowler *et al*. 1999), we believe that refactoring tool cannot be completely automated. A good refactoring tool should give the developers the final authority. It must interact with the developer because inferring design intent is difficult. Some refactorings may introduce a design change which requires software developer's insights because he has the best knowledge of the program context. A tool can facilitate the process by suggesting a set of refactorings and helps ensure that each refactoring is applied correctly.

3. Limitation and Scope of Research

A software system becomes harder to maintain as it evolves over a period of time. Its design becomes more complicated and difficult to understand; hence it is necessary to reorganize the code once in a while. The most important thing when reorganizing code is to make sure that the program behaves the same way as it did before the reorganization has taken place. Semantic preserving program transformations are known as refactorings. The idea of refactoring is First introduced by (William Opdyke *et al*. 1992). The behavior preservation criterion is also discussed in his work.

In the past, refactorings were not taken into the mainstream development process because applying refactorings by hand is error-prone and time consuming. The benefits of refactorings are not obvious to many developers because refactoring neither adds new features to the software nor improves any external software qualities. Therefore, many system developers give refactorings low priority. They are afraid that doing so would slow down the process and/or break their working code. Though refactoring does not help improve external software qualities, it helps improve internal software qualities such as reusability, maintainability and readability. It is inarguable that software design changes frequently during the development. Performing refactoring introduces a good coding discipline as it encourages reuse of existing code rather than rewriting new code from scratch. Refactoring is usually initiated/invoked by the developer. Most software developers only refactor their code when it is really necessary because this process requires in-depth knowledge of the software system. While many experienced developers can recognize the pattern and know when to refactor, novice programmers may find this process very difficult. Even with the knowledge of refactorings, it is not easy for the developer to determine which part of their code can benefit from refactorings. Many programmers learn from their experience. New generation programmers are more fortunate since Martin Fowler and Kent Beck address this issue in their book on Refactoring (Opdyke and Fowler *et al*. 1999). They provide a list of troubled code patterns which could be alleviated by refactorings. Such patterns are widely known as code smells or bad smells (Satwinder Singh *et al*. 2011, N. Maneerat *et al*. 2011). Recent work by Mantyla and others (Mika Mantyla, *et al*. 2003), attempts to make Fowler's long monotonous list of smells more understandable. In their work, smells are classified into 7 different categories. The taxonomy helps recognize relationships between smells and make them more comprehensible to the developer. Despite the presence of such guidelines, finding code smells is not trivial. First and foremost, the developer has to recognize those patterns. The problem is, even if he can recognize them, he may not realize it when he finds one. Such a task becomes much more difficult for

a large scale software system. The process of detecting and removing code smells with refactorings can be overwhelming. Without experience and knowledge of the design of the particular software, the risks of breaking the code and making the design (N. Maneerat *et al*. 2011) worse are high. Applying refactoring carelessly can inadvertently change program behavior. When refactoring is carefully applied, we not only preserve the program behavior but also avoid introducing new bugs. Many refactoring tools have been developed (James M. Bieman *et al*. 1998, Twan van Enckevort *et al*. 2009, Kelvin H. T. Choi *et al*. 2007, sourceforge.net.JRefactory 2003, eclipse.org. Eclipse 2003). There are also a number of works on finding refactoring candidates (YoshioKataoka *et al*.2001 , Frank Simon *et al*. 2001, Serge Demeyer *et al*. 2000). Nonetheless, these two frameworks usually work separately. It is unfortunate to see two related frameworks work on their own and not utilize the benefits to their maximum potentials. The relationship between code smells and refactorings are obvious but not many people have put them together.

4. Aim and Objectives of Research

Aim

To consider bad code smell as a measure of software maintainability after removing it, which will improve the software more maintainable.

Objectives

1.To ensure behavior preservation before refactoring bad code smell.
2.To find and remove duplicate code from application.
3.To enhance readability of generated code.
4.To enhance the scope of the application.
5.To remove dead code within application.
6.To improve the efficiency of overall system by increasing execution time and by decreasing software maintenance cost.

5. Methodology

5.1 Existing Components

There are two main components in this work: Fluid and Eclipse. The Fluid infrastructure will be used mainly for program analysis and code transformations. The Eclipse framework will be used as a front-end that interacts with the user.

5.1.1 Fluid

Fluid provides a tool to assure that the program follows the programmer's design intent. The developers can run different analyses on their programs. There are a number of analyses that this work uses which include Effects Analysis and Uniqueness Analysis. The analysis framework is set up in a way that a new analysis can be added easily and without too much hassle. Program analysis is usually done on an intermediate form that represents the program's structure. The representations that are commonly used are graph and tree. Fluid provides tree-based analyses. The internal representation (IR) which will be used in Fluid consists of nodes and slots. A node can be used to represent many kinds of objects but this work refers to a node in an abstract syntax tree (AST). A slot can store a value or a reference to a node. A slot can be attached to a node by using an “attribute" or can be collected into a "container". For instance, a method declaration node has an attribute “name" that holds the name of the method and is associated with a container that holds references to other nodes (its children) in the AST e.g., a list of parameters, a return type and a method body. Every analysis in Fluid will be performing on the IR. However, since Fluid IR is an internal representation that the developers do not usually understand, we need a way to obtain the source code back from the IR. This process is called unparsing. After the transformations are performed on the IR i.e., refactorings, the unparser is used to obtain the source code which is then displayed to the user.

5.1.2 Eclipse

Our requirement is to develop a tool that works inside an IDE. We choose Eclipse because it is one of the most popular IDEs for Java. It is also easy to extend via a plug-in which eliminates the need to develop the whole user interface from scratch.

5.1.3 Incompatibilities between Fluid and Eclipse

There are a number of incompatibilities between Fluid and Eclipse which make the implementation difficult. One issue involves different representations of the Fluid Abstract Syntax Tree (FAST) and Eclipse Abstract Syntax Tree (EAST). The FAST is more fine-grained than the EAST. On the Fluid's side, the Java source adapter has been implemented to address such a conflict. The Java source adapter, as its name implies, adapts the EAST into the FAST. It basically converts the abstract syntax tree obtained from Eclipse into a different abstract syntax tree with Fluid IR. The adapter allows us to perform different analyses on the Fluid side, since all analyses expect the FAST. The other and more serious issue is concerned with versioning. While Fluid is versioned, Eclipse is not. In other words, Fluid keeps track of changes made in different versions but there is no versioning system in Eclipse. To make them work together we need a mapping mechanism from non-version to version space and vice versa. Eclipse has no knowledge of which system's version (under the Fluid's context) it is working on. Hence, we need a bridge to administer the communication between Eclipse and Fluid. The bridge handles everything that involves Fluid versioning system. Its main duty is to keep Fluid and Eclipse synchronized on resource changes.

6. Implications

This work proposes and develops a framework that combines the processes of identifying and applying refactorings. Such a framework allows information from analysis to be reused and avoid recomposing any information. Therefore, it helps to improve the efficiency of the overall system. This work shows that integrating the two processes give us more satisfactory results. It forces us to look at the problem in a big picture. Program analysis plays a crucial part in this work.

However, it is easier and far more reliable if refactoring is implemented using robust tools that can automate the desired changes. Refactoring clarifies your code into a more efficient form. By transforming your code with refactoring techniques it will be faster to change, execute, and download. Refactoring is an excellent best practice to adopt for programmers wanting to improve their productivity

References

Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black(2012), How We Refactor, and How We Know It, in *IEEE transactions on software engineering*, vol. 38, NO.

Satwinder Singh and K. S. Kahlon(2011), Effectiveness of encapsulation and object oriented metrics to refactor code and identify error prone classes using bad smell, in *SIGSOFT Software Engineering*, 36(5):1-1

N. Maneerat and P. Muenchaisri(2011), Bad-smell prediction from software design model using machine learning techniques, in *Journal Computer Science and Software Engineering (JCSSE)*, *2011 Eighth International Joint Conference* on, pages 331-336

Katsuhisa Maruyama and Takayuki Omori(2011), A security-aware refactoring tool for Java programs in Refactoring Tools, *WRT '11 ACM, New York, USA*, pages 22-2

Frank Tip, Robert M. Fuhrer, Adam Kiezun, Michael D. Ernst, IttaiBalaban, and Bjorn De Sutter(2011), Refactoring using type constraint" in *ACM Transaction Programming Language System*, 33(3):9:1-9:4

Naouel Moha, Yann-Gael Gueheneuc, Laurence Duchien, and Anne-Francoise Lemeur(2010), Decor: A method for the specification and detection of code and design smells, *IEEE Transactions on Software Engineering*, volume 36, pages 20-36

E. Murphy-Hill and A.P. Black(2008), Refactoring Tools: Fitness for Purpose, *IEEE Software*, vol. 25, no. 5, pp. 38-44

G.C. Murphy, M. Kersten, and L. Findlater(2006), How Are Java Software Developers Using the Eclipse IDE?, *IEEE Software*, vol. 23, no. 4,pp. 76–83

Tom Mens, Tom Tourwe(2004), A Survey of Software Refactoring, in *IEEE Transactions on Software Engineering*, Vol. 3, No. 2, 126-1

K.J. Lieberherr and I.M. Holland(1989), Assuring good style for object-oriented programs, *IEEE Software*, vol.6, no.5, pp. 38-

GabrieleBavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto(2010), A two step technique for extract class refactoring, in *IEEE/ACM international conference on Automated Software Engineering, ASE '10*, pages 151-154

Twan van Enckevort(2009), Refactoring UML models: using open architecture ware to measure UML model quality and perform pattern matching on UML models with OCL queries, in *conference companion on Object oriented programming systems languages and applications, OOPSLA '09 ACM, New York, NY, USA*, pages 635-646.

EmmadSaadeh and Derrick G. Kourie(2009), Composite refactoring using fine-grained transformations*, in Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists (SAICSIT '09) ACM, New York, NY, USA*, pages 22-29

EmmadSaadeh, Derrick Kourie, and Andrew Boake(2009), Fine-grain transformations to refactor UML model, in WUP '09 ACM, New York, NY, USA, pages 45-

Kelvin H. T. Choi and Ewan Tempero (2007), Dynamic measurement of polymorphism, in *Thirtieth Australasian conference on Computer science -Volume 62, ACSC '07 Australian Computer Society, Inc., Darlinghurst, Australia, Australia*, pages 211-2

Danny Dig, Can Comertoglu, DarkoMarinov, and Ralph Johnson(2006), Automated detection of refactorings in evolving components, in *20th European conference on Object-Oriented Programming, ECOOP'06 Springer-Verlag, Berlin, Heidelberg*, pages 404-428

Huiqing Li, Claus Reinke, and Simon Thompson(2003), Tool support for refactoring functional program, in *ACM SIGPLAN Workshop on Haskell* , pages 27-

Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer (2003), Towards automating source-consistent UML refactorings, in *Sixth International Conference on the Unified Modeling Langua*

MikaMantyla, JariVanhanen, and Casper Lassenius(2003), A taxonomy and an initial empirical study of bad smells in code, in *IEEE International Conference on Software Engineering (ICSE '03)*, pages 381-38

YoshioKataoka, Michael D. Ernst, William G. Griswold, and David Notkin(2001), Automated support for program refactoring using invariants, in *International Conference on Software Maintenance (ICSM '01), Florence, Italy, November 6-10 IEEE Computer Society, Los Alamitos, California*, pages 736-74

Frank Simon, Frank Steinbruckner, and Claus Lewerentz(2001), Metrics based refactoring, in *5th European Conference on Software Maintenance and Reengineering (CSMR '01), Lisbon, Portugal, March 14-16, IEEE Computer Society, Los Alamitos, California* pages 30-38..

W. G. Griswold, D. Notkin(1993), Automated assistance for program restructuring - *ACM Trans. Software Engineering and Methodology*, 2(3): 228-26

P. L. Bergstein(1997), Maintenance of Object-Oriented Systems during Structural Evolution - *TAPOS Journal* 3(3): 185-21

D. Roberts, J. Brant, R. E. Johnson(1997), A refactoring tool for Smalltalk – *TAPOS '97 Journal of Theory and Practice of Object Systems*, 3(4): 253-26

William G. Griswold and Robert W. Bowdidge(1993), Program restructuring via design-level manipulation, in *IEEE International Conference on Software Engineering (ICSE '93), Baltimore, Maryland, USA*, pages 127-13

Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts(1999), Refactoring: Improving the Design of Existing Code, *Addison Wesley Longman, Reading, Massachussetts, USA*

James M. Bieman and Byung-Kyoo Kang(1998), Measuring design-level cohesion, *Software Engineering*, 24(2):111-12

Serge Demeyer, StephaneDucasse, and Oscar Nierstrasz(2000),Finding refactorings via change metrics, *In OOPSLA'00 Conference Proceedings Object-Oriented Programming Systems, Languages and Applications, Minneapolis, Minnesota, USA, October 15-19, ACM SIGPLAN Notices*, 35(10):166-178

W. G. Griswold(1991), Program restructuring as an aid to software maintenance - *University of Washington, USA*.

W. F. Opdyke(1992), Refactoring object-oriented frameworks – *PhD thesis University of Illinois at Urbana-Champaign, U*

D.Roberts(1999), Practical Analysis for Refactoring- *University of Illinois at Urbana-Champ*

S. Tichelaar(2001), Modeling object-oriented software for reverse engineering and refactoring, *University of Bern, Switzerla*

IntelliJ. IDEA(2002), *http://www.intellij.com/idea*

Aqris. RefactorIt(2001), *http://www.refactorit.co*

sourceforge.net.JRefactory(2003), *http://jrefactory.sourceforge.ne*

Instantiations.jFactor(2002), *http://www.instantiations.com/jfactor.*

sourceforge.net.Transmogrify(2001), *http://transmogrify.sourceforge.net*

eclipse.org. Eclipse(2003), *http://www.eclipse.org*

XRef-Tech. XRefactory(1998) *http://www.xref-tech.com/spell*

sourceforge.net.CppRefactory(2001), *http://cpptool.sourceforge.net.*