Enhanced semi-Automated Refactoring Engine with Behavioral testing

S.Ayshwaryalakshmi
Assistant Professor-Department of CSE,
University College of Engineering-Panruti,

Kannan M
M.E.(Computer Science and Engineering)
University College of Engineering-Panruti

***Abstract*— Refactoring is a transformation that preserves the external behavior of a program and improves its internal quality. Usually, compilation errors and behavioral changes are avoided by preconditions determined for each refactoring transformation. However, to formally define these preconditions and transfer them to program checks is a rather complex task. In practice, refactoring engine developers commonly implement refactoring in an ad hoc manner since no guidelines are available for evaluating the correctness of refactoring implementations. As a result, even mainstream refactoring engines contain critical bugs. The system technique presents a technique to test Java refactoring engines. It automates test input generation by using a Java program generator that exhaustively generates programs for a given scope of Java declarations. The refactoring under test is applied to each generated program. The technique uses SAFEREFACTOR, a tool for detecting behavioral changes, as an oracle to evaluate the correctness of these transformations. Finally, the technique classifies the failing transformations by the kind of behavioral change or compilation error introduced by them. The main objective of this paper is to create the Java Integrated Development Environment with Refactoring Engine and automated Behavioral Testing process.**

I. INTRODUCTION

Software has become integral part of most of the fields of human life. We name a field and we find the usage of software in that field. Software applications are grouped in to eight areas for convenience System software: Infrastructure software come under this category like compilers, operating systems, editors, drivers, etc. Basically system software is a collection of programs to provide service to other programs. Real time software is used to monitor, control and analyze real world events as they occur. An example may be software required for weather forecasting. Such software will gather and process the status of temperature, humidity and other environmental parameters to forecast the weather. Embedded software is placed in Read-Only-Memory of the product and controls the various functions of the product. The product could be an aircraft, automobile, security system, signaling system, control unit of power plants, etc. The embedded software handles hardware components and is also termed as intelligent software. Business software is the largest application area. The software designed to process business applications is called business software. Business software could be payroll, file monitoring system, employee management, and account management. It may also be a data warehousing tool which helps us to take decisions based on available data. Management information system, enterprise resource planning and such other software are popular examples of business software. Personal computer software is used in personal computers is covered in this category. Examples are word processors, computer graphics, multimedia and animating tools, database management, computer games etc. This is a very upcoming area and many big organizations are concentrating their effort here due to large customer base. Artificial Intelligence software makes use of non- numerical algorithms to solve complex problems that are not amenable to computation or straight forward analysis. Examples are expert systems, artificial neural network, signal processing software etc. Web based software: The software related to web applications comes under this category. Scientific and engineering application software is grouped in Engineering and scientific software. Huge computing is normally required to process data.

Software products represent the information-intensive artifacts that are incrementally constructed and iteratively revised through a software development effort. Such efforts can be modeled using software product life cycle models. These product development models represent an evolutionary revision to the traditional software life cycle models. The revisions arose due to the availability of new software development technologies such as software prototyping languages and environments, reusable software, application generators, and documentation support environments. Each of these technologies seeks to enable the creation of executable software implementations either earlier in the software development effort or more rapidly. Therefore in this regard, the models of software development may be implicit in the use of the technology, rather than explicitly articulated. This is possible because such models become increasingly intuitive to those developers whose favorable experiences with these technologies substantiate their use. Thus, detailed examination of these models is most appropriate when such technologies are available for use or experimentation.

II. REFACTORING ENGINE:

Software refactoring is used to restructure the internal structure of object-oriented software to improve its quality, especially its maintainability, extensibility, and reusability, while preserving its external behavior. Software refactoring is widely used to delay the degradation effects of software aging and facilitate software maintenance. Because software is repeatedly modified according to evolving requirements, source code shifts from its original design structure. The source code becomes complex, difficult to read or debug, and

even harder to extend. Software refactoring improves readability and extensibility by cleaning up Bad Smell (unstructured program)s in the source code.
A key issue in software refactoring is determining the kind of source code that requires refactoring. Experts have summarized typical situations that may require refactoring.

A. Evaluated Bad Smell (unstructured program)s:

As an initial study, we consider nine kinds of Bad Smell (unstructured program)s for evaluation. A brief introduction is presented here so that the paper can be understood on its own.
a. Duplicated Code. Duplicated Code is comprised of fragments of source code appearing in more than one location. In a narrow sense, only identical copies are called Duplicated Codes. In a broad sense, however, they may include slightly different fragments resulting from copy-paste-modify editions.
b. Long Method. The longer a method is, the harder it is to read or modify. Consequently, long and complex methods should be divided into short and well-named methods with refactoring rules.
c. Large Class. Large classes usually try to take too many responsibilities, making them complex and confusing. To improve their readability and maintainability, large classes should be divided into smaller ones, each for a single responsibility.
d. Long Parameter List. Methods with too many parameters are difficult to use and even harder to change. The parameters can ordinarily be replaced with a few objects that contain the parameters.
e. Feature Envy. A method or a fragment of a method may be more interested in features of another class than those of the enclosing class, which is called feature envy.
f. Primitive Obsession. Primitive Obsession is the situation in which objects should have been used instead of primitives. It is further divided into three subcategories: Simple Primitive Obsession, Simple Type Code, and Complex Type code. The division is necessary because different refactoring rules should be applied depending on whether the primitive object is a type code and whether the type code influences the behavior of the enclosing class. If the primitive object is not a type code, the refactoring is simple: The data value is replaced with an object. If it is a type code, the refactoring is more complicated, depending on whether the type code affects behavior. If it does, refactoring rule Replace Type Code with Subclasses should be applied. Otherwise, Replace Type Code with Class is preferred.
f. Useless Field. It is a synonym of Dead Field, referring to fields defined but never used.
g. Useless Method. Once a domain class is found, designers may propose methods (operations) for it. Unfortunately, sometimes certain operations are irrelevant to the role the class plays in the specific system. These operations are useless in the system.
h. Useless Class. Useless classes are those defined but never used. Typically, these are results of inappropriate boundaries of systems.

B. Refactoring Activities:

The refactoring process consists of a number of distinct activities:

a. Identify where the software should be refactored.
b. Determine which refactoring should be applied to the identified places.
c. Guarantee that the applied refactoring preserves behavior.
d. Apply the refactoring.
e. Assess the effect of the refactoring on quality characteristics of the software e.g., complexity, understandability, maintainability or the process e.g., productivity, cost, effort.
f. Maintain the consistency between the refactored program code and other software artifacts such as documentation, design documents, requirements specifications, tests, etc.

III. OVERVIEW OF BEHAVIORAL TESTING:

The software developers use an automatic approach to classify compilation errors. It consists of splitting the failing tests based on messages from the test oracle. The goal is to group together the failing tests related to the same bug. The traditional approach ignores (package, class, method, or field) names within quotes. If the same refactoring is applied to two different programs, and they result in compilation error messages following the same template, a single bug is assigned to these two failures. We developed a tool to automate this grouping.

Additionally, they propose an approach to classify behavioral changes by analyzing each detected change based on the characteristics of each pair source program-target program. Our approach is based on a set of filters; a filter checks whether the programs follow a specific structural pattern. For example, there are filters for transformations that enable or disable overloading/overriding of a method in the target program, relatively to the source program. All filters are presented in Table 1.3. We defined these filters by analyzing bugs found through the use of our approach, in addition to other bug reports from refactoring engines. The filters may be applied in any order. The bug category of a behavior-changing transformation is then designated by the filters matched by its source and target programs. When a transformation does not fit any of these filters, conventional debugging is demanded from refactoring engine developers. For instance, the failure in the Pull up Method on either Eclipse JDT 3.7 or JRRTv1 matches the filter named “Changes super (this) to this (super)” from Table 1.3, in which a problem with replacing a reference to super with this is detected. The set of filters is not complete. New filters can be proposed based on additional bugs found by refactoring engine developers. Currently, the classification of behavioral changing transformations is carried out manually. The process consists of analyzing each pair of programs and testing every filter for matches.

Table 1.3 Filters for classifying Behavioral Changes

Filter	Description
Enables / disables overriding	A Method comes to be overridden, After refactoring

Enables / disables overloading	A Method comes to be overloaded, After refactoring
Enables/disables field hiding	A field comes to be hidden by another filed declaration, After refactoring
Shadows class declaration	A class declaration comes to be shadowed by another declaration
Maintains Super while changing hierarchy	A reference to super is moved up or down the hierarchy during refactoring
Changes Accessibility	The refactoring changes the access modifier of a given filed or method
The refactoring program crashes	The original program is normally executed by the test suite but the refactoring program one throws some exceptions
Enables / Disables implicit cast	An implicit cast between primitive types is applied where it did not take place originally

IV. EXISTING TECHNIQUES:

A. *Semi Automated Refactoring Engine with Regression Testing*

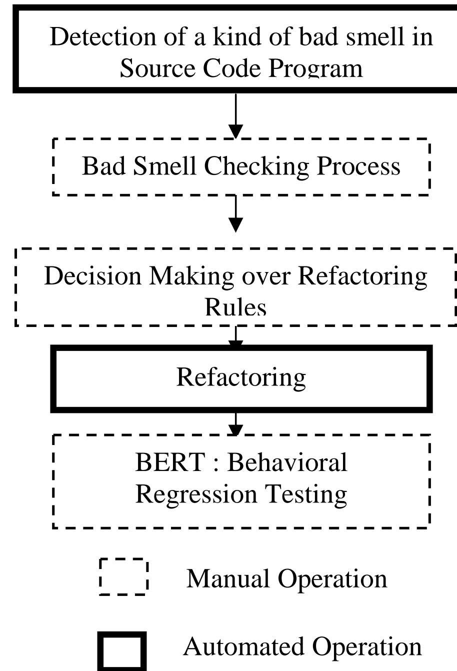
A key issue in software refactoring is determining the kind of source code that requires refactoring. Experts have summarized typical situations that may require refactoring. Fowler et al. call them Bad Smell (unstructured program)s, indicating that some part of the source code smells terrible. In other words, Bad Smell (unstructured program)s (e.g., Duplicated Code) are signs of potential problems in code that may require refactoring. The definition and explanation of Bad Smell (unstructured program)s can be found in the third chapter of their book. These Bad Smell (unstructured program)s are usually linked to corresponding refactoring rules that can help dispel these Bad Smell (unstructured program)s.

First, uncovering Bad Smell (unstructured program)s 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 Smell (unstructured program)s 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 Smell (unstructured program)s. Manual refactoring is time-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 seek to address.

Existing System Architecture:

The detection tool proposes initial results that require manual confirmation. Once the detected Bad Smell (unstructured program) is confirmed, the software engineer decides how to refactor it. Selected refactoring rules are manually or semi-automatically applied to the Bad Smell (unstructured program)s with the help of refactoring tools.



Then, the software engineer moves on to the next kind of Bad Smell (unstructured program)s, and repeats the process until all kinds of Bad Smell (unstructured program)s have been detected and resolved.

As a result, different kinds of Bad Smell (unstructured program)s are detected and resolved one after the other, regardless of whether the sequence is arranged consciously or unconsciously.

In earlier the detection of Bad Smell (unstructured program)s in the code remains automated but the checking process of the Bad Smell (unstructured program)s and to determine how to restructure Bad Smell (unstructured program)s in terms of refactoring rules that should be applied, and arguments of the rules may also be semi-automated, it need human intervention.

Most Bad Smell (unstructured program)s automatically detected should be rechecked manually because 100 percent precision cannot be guaranteed by detection tools.

- In earlier the detection of Bad Smell (unstructured program)s in the code remains automated but the checking process of the Bad Smell (unstructured program)s and to determine how to restructure Bad Smell (unstructured program)s in terms of refactoring rules that

should be applied, and arguments of the rules may also be semi-automated, it need human intervention.

- Most Bad Smell (unstructured program)s automatically detected should be rechecked manually because 100 percent precision cannot be guaranteed by detection tools.

V. PROBLEM STATEMENT:

Bad Smell (unstructured program)s are signs of potential problems in code. Detecting and resolving Bad Smell (unstructured program)s, however, remain time-consuming for software engineers despite proposals on Bad Smell (unstructured program) detection and refactoring tools. Numerous Bad Smell (unstructured program)s have been recognized, yet the sequences in which the detection and decree of different kinds of Bad Smell (unstructured program)s are performed are rarely discussed because software engineers do not know how to optimize sequences or determine the benefits of an optimal sequence. The behavioral changes in the restructures code may remain make error in compilation or in run time. Thus the changes should be tested as automatically.

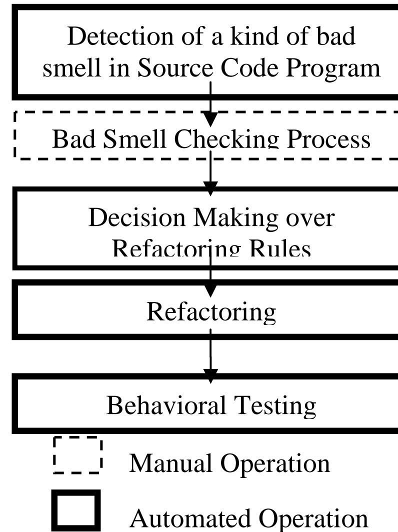
VI. PROPOSED WORK:

Usually, compilation errors and behavioral changes are avoided by preconditions determined for each refactoring transformation. However, to formally define these preconditions and transfer them to program checks is a rather complex task. Bad Smell (unstructured program)s are signs of potential problems in code. Detecting and resolving Bad Smell (unstructured program)s, however, remain time-consuming for software engineers despite proposals on Bad Smell (unstructured program) detection and refactoring tools. Numerous Bad Smell (unstructured program)s have been recognized, yet the sequences in which the detection and decree of different kinds of Bad Smell (unstructured program)s are performed are rarely discussed because software engineers do not know how to optimize sequences or determine the benefits of an optimal sequence. The behavioral changes in the restructures code may remain make error in compilation or in run time. Thus the changes should be tested as automatically.

In practice, refactoring engine developers commonly implement refactoring in an ad hoc manner since no guidelines are available for evaluating the correctness of refactoring implementations. As a result, even mainstream refactoring engines contain critical bugs. The system technique presents a technique to test Java refactoring engines. It automates test input generation by using a Java program generator that exhaustively generates programs for a given scope of Java declarations. The refactoring under test is applied to each generated program. The technique uses SAFEREFACTOR, a tool for detecting behavioral changes, as an oracle to evaluate the correctness of these transformations. Finally, the technique classifies the failing transformations by the kind of behavioral change or compilation error introduced by them.

The contributions of this paper are as follows: First, it discovers and illustrates the importance of decree sequences of Bad Smell (unstructured program)s. Second, it proposes a decree sequence for commonly occurring Bad Smell (unstructured program)s. Finally, it validates the effect of decree sequences of Bad Smell (unstructured program)s on two nontrivial applications.

System Architecture:



The system checks for compilation errors in the resulting program and reports those errors; if no errors are found, it analyzes the results and generates a number of tests suited for detecting behavioral changes.

This approach is based on a set of filters; a filter checks whether the programs follow a specific structural pattern. For example, there are filters for transformations that enable or disable overloading/overriding of a method in the target program, relatively to the source program. Thus this system helps to overcome the issues in existing system.

A. Semi Automated Refactoring Engine:

Refactoring is a transformation that preserves the external behavior of a program and improves its internal quality. Usually, compilation errors and behavioral changes are avoided by preconditions determined for each refactoring transformation. However, to formally define these preconditions and transfer them to program checks is a rather complex task. In practice, refactoring engine developers commonly implement refactorings in an ad hoc manner since no guidelines are available for evaluating the correctness of refactoring implementations. As a result, even mainstream refactoring engines contain critical bugs. Most Bad Smell (unstructured program)s automatically detected should be rechecked manually because 100 percent precision cannot be guaranteed by detection tools.

The process of make decisions over the determination of refactoring rules to resolve the Bad Smell (unstructured program)s can be optimized as automated that it is possible to analyze the impact of Bad Smell (unstructured program)s by analyzing historical information. Thus the Bad Smell (unstructured program)s can be restructured by refactoring process.

Automated refactoring Engine:

Additionally, the system technique proposes an approach to classify behavioral changes by analyzing each detected change based on the characteristics of each pair source program-target program. Our approach is based on a set of filters; a filter checks whether the programs follow a specific structural pattern. For example, there are filters for transformations that enable or disable overloading/overriding of a method in the target program, relatively to the source program. We defined these filters by analyzing bugs found through the use of our approach, in addition to other bug reports from refactoring engines. The set of filters is not complete. Currently, they focus on the Java constructs supported by java editor refactoring engine. New filters can be proposed based on additional bugs found by refactoring engine developers. Currently, the classification of behavioral changing transformations is carried out manually. The process consists of analyzing each pair of programs and testing every filter for matches.

VII.ADVANTAGES

- The contributions of this paper are as follows: First, it discovers and illustrates the importance of decree sequences of Bad Smell (unstructured program)s. Second, it proposes a decree sequence for commonly occurring Bad Smell (unstructured program)s. Finally, it validates the effect of decree sequences of Bad Smell (unstructured program)s on two nontrivial applications.
- The system checks for compilation errors in the resulting program and reports those errors; if no errors are found, it analyzes the results and generates a number of tests suited for detecting behavioral changes.
- This approach is based on a set of filters; a filter checks whether the programs follow a specific structural pattern. For example, there are filters for transformations that enable or disable overloading/overriding of a method in the target program, relatively to the source program.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we first motivated the necessity of arranging decree sequences of Bad Smell (unstructured program)s with an example. Then, we illustrated how to arrange such a decree sequence for commonly occurring Bad Smell (unstructured program)s. We also carried out evaluations on two nontrivial applications to validate the research. The results suggest a significant reduction in refactoring can be achieved when Bad Smell (unstructured program)s are resolved using the recommended decree sequence. The contributions of this paper are as follows: First, it discovers and illustrates the importance of decree sequences of Bad Smell (unstructured program) s. Second, it proposes a decree sequence for commonly occurring Bad Smell (unstructured program)s. Third, it validates the effect of decree sequences of Bad Smell (unstructured program)s on two nontrivial applications. Finally, the technique classifies the failing transformations by kind of behavioral change or compilation error introduced by them. We propose a Java program generator to run the program generation step of our technique.

The contributions of this paper are as follows: First, it discovers and illustrates the importance of decree sequences of Bad Smell (unstructured program)s. Second, it proposes a decree sequence for commonly occurring Bad Smell (unstructured program)s. Finally, it validates the effect of decree sequences of Bad Smell (unstructured program)s on two nontrivial applications.

Thus the above project which deals with the application oriented languages. It has been enhanced with the Web Pipe like mashups. The mashup are becoming increasingly popular as end users are able to easily access, manipulate, and compose data from many web sources. We have observed, however, that mashups tend to suffer from deficiencies that propagate as mashups are reused. To address these deficiencies, we would like to bring some of the benefits of software engineering techniques to the end users creating these programs. In this work, we focus on identifying code smells indicative of the deficiencies we observed in web mashups programmed in the popular Yahoo! Pipes environment.

IX. REFERENCES:

[1]B. Daniel, D. Dig, K. Garcia, and D. Marinov, “Automated Testing of Refactoring Engines,” Proc. Sixth Joint Meeting European Software Eng. Conf. and ACM SIGSOFT Symp. The Foundations of Software Eng., pp. 185-194, 2007.

[2]B. Korel and A.M. Al-Yami, “Automated Regression Test Generation,” Proc. Fourth Int’l Symp. Software Testing and Analysis, pp. 143-152, 1998.

[3]F. Steimann and A. Thies, “From Public to Private to Absent: Refactoring Java Programs under Constrained Accessibility,” Proc. 23rd European Conf.Object-Oriented Programming, pp. 419-443, 2009.

[4]G.Soares et al., “A Survey of Software Refactoring,” IEEE Trans. Software Eng., vol. 39, no. 2, pp. 147-162, Feb. 2013.

[5]H. Li and S. Thompson, “Testing Erlang Refactorings with QuickCheck,” Proc. 19th Int’l Symp. Implementation and Application of Functional Languages, pp. 19-36, 2008.

[6]T. Mens and T. Tourwe ´, “A Survey of Software Refactoring,” IEEE Trans. Software Eng., vol. 30, no. 2, pp. 126-139, Feb. 2004.

[7]T. Mens, S. Demeyer, and D. Janssens, “Formalising Behaviour Preserving Program Transformations,” Proc. First Int’l Conf. Graph Transformation, pp. 286-301, 2002.