Correctness of Semantic Code Smell Detection Tools

Neeraj Mathur* and Y Raghu Reddy†
*†Software Engineering Research Center,
International Institute of Information Technology, Hyderabad (IIIT-H), India
*neeraj.mathur@research.iiit.ac.in, †raghu.reddy@iiit.ac.in

***Abstract*—Refactoring is a set of techniques used to enhance the quality of code by restructuring existing code/design without changing its behavior. Refactoring tools can be used to detect specific code smells, propose relevant refactorings, and in some cases automate the refactoring process. However, usage of refactoring tools in industry is still relatively low. One of the major reasons being the veracity of the detected code smells, especially smells that aren't purely syntactic in nature. We conduct an empirical study on some refactoring tools and evaluate the correctness of the code smells they identify. We analyze the level of confidence users have on the code smells detected by the tools and discuss some issues with such tools.**

***Index Terms*—Correctness, Detection, Maintenance, Refactoring, Semantic code smells.**

I. Introduction

Refactoring improves various qualities of code/design like maintainability (understandability and readability), extensibility, etc. by changing the structure of the code/design without changing the overall behaviour of the system. It was first introduced by Opdyke and Johnson [11] and later popularized by Martin Fowler [7]. Fowler categorized various types of refactorings in terms of their applicability and suggested various refactorings for code smells (bad indicators in code) within classes and between classes. Over the years, other researchers have added to the knowledge base on code smells [16]. Any refactoring done to address specific code smells requires one to test the refactored code with respect to preservation of the original behaviour. This is primarily done by writing test cases before the refactoring is done and testing the refactored code against the test cases. As a result, validating the correctness of a detected code smell and the automation of it's refactoring is very difficult. Explicit manual intervention may be needed.

Many code smell detection tools support (semi) automatic refactoring process [4], [8]. *iPlasma, Jdeodorant, RefactorJ, etc*. are some of the tools that can be used for detection of code smells and application of specific refactoring techniques in automated or semi-automated manner. As noted in our previous work, each of these tools can detect only certain type of code smells automatically [8]. Also, there is no standardized approach for detecting such code smells and hence tools follow their own approach [9], inevitably leading to different set of code smells being detected for the same piece of code. Most code smell detection techniques depend on static analysis and code metrics and do not consider factors like *system size, language structure* and *context*. In other words, any design-based refactorings require the tool to understand the actual semantic intent of the code itself. For example, Long method is one such code smell that requires the tool to understand the context of the method before an Extract method refactoring can be performed automatically while preserving the original semantic intent.

Despite the known benefits of refactoring tools, their usage is not widespread due to users' lack of trust on the tools' code smell detection capability, learning curve involved, and the inability of users to understand the code smell detection results [4], [6], [14]. In this paper, we study the correctness of the detected code smells of multiple open source tools like JDeodorant, InCode, etc. and the lack of trust of users on their detection capability through an empirical study of open source projects like *JHotDraw (www.jhotdraw.org)* and *GanttProject (www.ganttproject.biz)*. We believe that lack of trust is proportional to the correctness of the tools detection ability. Most code smell detection tools detect code smells that require small-scale resolutions correctly. However, correctness is an issue when design based refactorings or semantic intent is considered. Hence, we restrict our focus to tools that detect code smells that require the tool to understand the semantic intent (for example, *Feature Envy, Long Methods, Shotgun Surgery, God class, etc.*). From now on, we refer to such code smells as "`semantic code smells`". We cross validate our study results on GanttProject with the study results conducted by Fontana et al. [6] on GanttProject (v1.11.1). Additionally, we used our own dummy code with a few induced semantic code smells to check for correctness of the tools. We focus on the following in this paper:

- Correctness of the identified code smells among the chosen tools
- Deviation in confidence levels of developers in open source code smell detection tools that detect semantic code smells

The rest of the paper is structured as follows: Section II provides a brief overview of the code smells discussed in this paper and section III presents some related work. In section IV, we detail the study design. Section V and VI present the results and analysis of the results. Based on the study, we provide some guidelines for increasing the correctness of detecting some code smells in section VII. Finally, in section VIII we discuss some limitations to our work.

II. Code Smells

Complexity related metrics like coupling are commonly used in tools to detect certain semantic code smells. Threshold values are established for various metrics and the code smells are identified based on the threshold values. Code smells like Feature envy, long methods, god class, etc. are widely studied in the literature. In our study, we primarily target the following code smells:

- *Feature envy*: A method is more interested in some other class than the one in which it is defined.
- *Long methods*: Method that is too long (measured in terms of lines of code or other metrics), possibly leading to low cohesion and high coupling
- *God class*: A God class performs too much work on its own delegating only minor details to a set of trivial classes.
- *Large Class*: A class with too many instance variables or methods. It may become a God class.
- *Shotgun Surgery*: A change in one classes necessitates a change in many other classes
- *Refused Bequest*: A sub-class not using its inherited functionality

III. Related Work

Fontana et al. [6] showed the comparative analysis of code smells detected by various refactoring tools and their support of (semi) automatic refactoring. The study analyzes the differences between code smell detection tools. In our previous work [8], we analyzed various java based refactoring tools with respect to its usability and reasoned about the automation of various code smells.

Pinto et al. [14] investigated data from StackOverflow to find out the barriers for adoption of code detection tools. They listed the issues mentioned in the forum related to the adoption/usability issues users are talking about in the StackOverflow. Olbrich et al. [10] performed an empirical study for God Class and Brain Class to evaluate that detected smells are really smells. From their empirical study they have concluded that if the results are normalized with the size of the system then smell results will become opposite. In fact the detected smell classes were less likely for changes and errors.

Ferme et al. [5] conducted a study for God Class and Data Class to find out that all smells are the real smells. They have proposed filters to be used to reduce or refine the detection rules of these smells. This paper extends our previous work and complements the work done by other authors by considering semantic code smells. In [13], Palomba et al. studied developers perception of bad smells. Their study depicts gap between theory and practice, i.e., what is believed to be a problem (theory) and what is actually a problem (practice). Their study provide insights on characteristics of bad smells not yet explored sufficiently.

Ouni et al. [12] proposed search based refactoring approach to preserve domain semantics of a program when refactoring is decided/implemented automatically. They argued that refactoring might be syntactically correct, have the right behaviour, but model incorrectly the domain semantics.

IV. Study Design

The objective of our study is to analyze the correctness of tools relevant to semantic code smells by performing a study with human subjects with prior refactoring experience. Tools like *JDeodorant, PMD, InCode, iPlasma*, and *Stench Blossom*, and two large systems like *JHotDraw* and *GanttProject* that have been widely studied in refactoring research were considered. Choosing these tools and systems helped us in cross validating our work with prior work done by us [8] and other researchers using similar systems/tools.

Initially, 35 human subjects volunteered to be part of the study. The exact hypothesis of the study was not informed to the subject to avoid biasing the study results. The subjects were only informed of the specific tasks that needed to be done, i.e. to assess certain refactoring tools with respect to their ability to detect code smells based on a given criteria and fill in a template.

All the human subjects had varying levels of prior knowledge about code smells and tool based refactorings. However, they had not worked on the specific tools used for this study. The subjects had varied experience (13% subjects had 1-5 years of experience, 31% had 5-10 years of experience and rest had less than a year of experience). The detailed statistics of subjects is available at [3] . We asked them to focus on specific code smells like Feature Envy, Long Methods, Shotgun Surgery, God class, etc. and list down all these smells and record their comments/rationale for detecting these as code smells. The template [1] had a column that asked them the semantic meaning of the detected code smells. In addition, we asked them to provide reasoning for not agreeing with the refactorings detected by the tools used. The subjects were given a three-week period to perform the activity and fill in templates. After evaluation of the templates, results from 32 subjects were taken into consideration. The other three did not fill in the templates completely. To cross-check the correctness of the tools with respect to their semantic code smell detection, in addition to the two open-source systems, we instrumented one of our own projects. It was interesting to note that most of the tools were not detecting code smells that seemed obvious from our perspective.

A. Subject Systems

In our study, we chose two open source systems:

- GanttProject - a free project management app where users can create tasks, create project baseline, organize tasks in a work break down structure
- JHotDraw - a Java GUI framework for technical and structured Graphics. Its design relies heavily on some well-known design patterns.

The subject systems are available for use under open source license and are fairly well documented. In addition to being a part of Qualitas Corpus [15], these are widely studied in refactoring research and referenced in the literature. Table 1 provides details of these systems.

TABLE I: Characteristics of subject systems

	JHotDraw	Ganttproject
Version	5.4	2.7.1891
Total Lines of Code	32435	46698
Number of classes	368	1230
Number of methods	3341	5917
Weighted methods per class	6990	8540
Number of static methods	280	235

B. Code Smell Detection Tools

There are several commercial and open source code smell detection tools. Some are research prototypes meant for detection of specific code smells while others identify a wide range of code smells. In addition to detection, some tools refactor the code smells automatically, while others are semi-automated. Semi-automated refactoring tools propose refactorings that require human intervention and can be automated to a certain extent by changing rules of detection. Some of these are more or less like recommender systems that recommend certain type of refactorings but leave it to the developers to refactor or ignore the recommendations.

Some tools are integrated with the IDE itself, while others are written as plugins that can be installed on a need basis. For example, checkstyle, PMD, etc. are some eclipse based plugins that are well known. For our study, we focused on tools widely studied in the literature, their semantic code smell detection ability and usage in the industry. The list of tools and the detected code smells relevant to our study are shown in table II.

V. Experiment Results

Table III provides a cumulative summary of code smells detected and disagreements (weighted average of results from 32 results) by our human subjects. To show the disparity in results, we compare our results for the GanttProject with Fontana et al.'s results [6]. For reference, the detailed list of detected smell by our human subjects is available at [2].

Feature Envy: The number of Feature Envy methods detected by different tools varies significantly. Some tools consider any three or more calls to method of the other class as a code smell and hence give rise to large number of false positive code smells. In such cases, when it's just counting numbers, it becomes tedious to filter out the actual smells.

The degree of disagreement was found to be 9/44 for JDeodorant and 3/4 for inCode for the JHotDraw project. Disagreements by the human subjects were prevalent across all tools for detected feature envy smells. We also observed a significant difference between the results of Fontana's [6] study and our JDeodorant results. Their study reveals that over a period of time, in GanttProject version from V1.10 to 1.11.1, Feature Envy methods reduced to 2 (in v1.11.1) from 18 (in v1.10), whereas in our study of GanttProject v2.7, there were 113 Feature Envy code smells. As it can be seen from the results, the number of detected smells is significantly different from the numbers given in their work.

God Class: The number of detected God classes detected by JDeodorant for GanttProject in Fontana et al.'s [6] study was 22 (v1.11.1) where as in v2.7 it is 127. In inCode the number of god classes were significantly lesser: reduced from 13 (v1.11.1) to 4 (v2.7). Unlike JDeodrant and inCode, the results from iPlasma increased: from 13 (v1.11.1) to 42 (v2.7). This inconsistency between tools reduces the confidence level of the results.

The degree of disagreement to jDeodrant code smells for jHotDraw project was 10 out of 56. Our human subjects observed that tools were considering the data model classes (getters and setters) and parser classes as smells. Usually these classes are needed and are necessary evils. As a result, there is a need of building some sort of intelligence/metrics to detect these kinds of classes which can safely be ignored to reduce the false positives.

Large Class: This code smell is detected by the code size and is subjective to the threshold limit of LOC set by the user. Classes that contain project utility methods can grow in size with a lot of small utility methods and usually developers make these classes as singleton objects. Refactoring such smells requires one to be able understand the intent of the sentences before an *extract method* or *extract class* refactoring is applied.

Refused Bequest: Some tools considered interface methods and abstract methods that are normally overridden by the respective concrete classes as a code smell resulting in a lot of false positives.

Compared to Fontana et al.'s [6] study, inCode detected 6 (in V2.7) whereas it was 0 (in V1.11.1). The degree of disagreement to the detected code smells was 2 in inCode, 1 in iPlasma for GanttProject. For jHotDraw it was 1 (for inCode) and 2 (for iPlasma).

Shotgun Surgery: The results of our study for shotgun surgery on v2.7 were similar to the one's for the GanttProject (v1.11.1). However, from our study we can conclude that the way shotgun surgery is detected in the different code smells differs from tool to tool rather that different versions of the same tool. The degree of disagreement to the detected code smells was 2 for GanttProject using inCode and 7 using iPlasma.

Long Method: This code smell is related to number of lines of code present in the method and subject to the threshold limit set by the user for detecting the code smell. As per the disagreements documented by our subjects, methods containing large switch statements should not be counted as a long method. iPlasma had a complex detection mechanism that considers such kind of things while detecting the Long Method code smells. On the contrary jDeodrant listed fairly small methods as a long method code smell.

The degree of disagreement to the code smells detected for GanttProject was 12 in jDeodrant, 27 in Stench Blossom and 8 in PMD. For JHotDraw project, it was 10 in jDeodrant, 12 in Stench Blossom, 11 in iPlasma and 8 in PMD.

The detections from jDeodrant tool were significantly high as compared to 57 (v1.111.1) of GanttProject from Fontana et al. [6] study. Interestingly, we noticed that the long methods were reduced from 160 (in v1.10) to 57 (in v1.11.1), whereas as in our study it was still high i.e. 221.This was because as a software evolves, it is expected that long methods will grow over a period of time.

TABLE II: Code smell detection tools

Tool	Code smells	Detail
JDeodarant, (vv. 3.5 e3.6), [EP],Java	FE, GC, LM, TC	This is an Eclipse plug-in that automatically identifies four code smells in Java programs. It ranks the refactoring according to their impact on the design and automatically applies the most effective refactoring.
Stench Blossom (v. 3.3), [EP],Java	FE, LM, LC, MC	This tool provides a high-level overview of the smells in their code. It is an Eclipse plugin with three different views that progressively offer more visualized information about the smells in the code.
InCode [SA], C, C++, Java	BM, FE, GC, IC, RB, SS	This tool supports the analysis of a system at architectural and code level.It allows for detection of more than 20 design flaws and code smells.
iPlasma [SA], C++, Java	BM, FE, GC, IC, SS, RB, LM, SG	It can be used for quality assessment of object-oriented systems and supports all phases of analysis: from model extraction up to high-level metrics based analysis, or detection of code duplication.
PMD, [EP or SA],Java	LC, LM	Scans Java source code and looks for potential bugs such as dead code, empty try/catch/finally/switch statements, unused local variables, parameters and duplicate code.
Feature Envy (FE), Refuse Bequest (RB), God Class (GC), Long Method (LM), Lazy Class (LC), Intensive Coupling (IC), Shotgun Surgery (SS), Speculative Generality (SG), Dispersed Coupling (DC), Brain Method (BM). Type: Standalone Application (SA), Eclipse Plug-in (EP)		

TABLE III: Code Smell Detected & Disagreement

Code Smell	jDeodrant		inCode		iPlasma		Stench Blossom		PMD	
	$	#	$	#	$	#	$	#	$	#
JHotDraw										
FE	44	9	4	3	35	4	28	10	56	9
GC	56	10	14	15	22	2	-	-	34	10
LC	-	-	-	-	-	-	22	5	41	5
RB	-	-	4	1	2	2	-	-	-	-
SS	-	-	10	3	13	6	-	-	-	-
LM	90	10	-	-	94	11	113	12	73	8
GnattProject										
FE	113	12	11	2	42	13	53	28	38	4
GC	127	8	4	3	24	3	-	-	37	17
LC	-	-	-	-	-	-	9	-	40	9
RB	-	-	6	2	3	1	-	-	-	-
SS	-	-	7	2	42	7	-	-	-	-
LM	221	12	-	-	-	-	55	27	36	8
$ - Detected, # - Disagree										

VI. Discussion

The major challenge in assessing correctness of detected smells is the knowledge possessed by human subjects in regards to the code smells and the behavior of the code itself. Since the subjects (users) were not familiar with the tools chosen for the study, they complained about the time consumed in understanding and using the tool. At times, the focus seemed to be more on the user-interface of the tool rather than detected code smells. The authors had to revert back to the subjects to get additional clarification about the comments written in the templates regarding specific code smells.

Some tools require explicit specification of rules (for example, PMD) for detecting a specific code smell. So, selecting rules from the entire list of rules was tedious and time-consuming. Most of the users seemed to struggle with the setup and configuration issues of the tools. Few tools like jDeodorant had high memory utilization and performance issues. So re-compilation after every change and re-running the detection process was laborious.

Contrary to our belief that the same code smell must be identified in the same way by different tools, the disagreement in the correctness of the detected code smells between the various tools for the same type of smell was pretty high (as shown in table III). Additionally, the results were not complimenting the results provided in prior research [6]. In other words, the correctness of the detected smells was not accurate with respect to the semantics of the code written. To validate the results, we had to further cross-check the tools with some dummy examples. For instance, the dummy code (shown in Listing 1) was detected as a *Feature Envy* smell in some of the tools (for example, *jDeodorant*). If '`doCommit`' method is moved to any of the classes (A, B and C in this example) we must pass the other two classes as a reference parameter, that in turn increases the coupling. Moreover, semantically it makes sense to call all commit methods in '`doCommit`' method itself.

An interesting observation can be made from the tools that detected *Request Bequest* code smell instances. For example, code resembling the dummy code (shown in Listing 2) reveals that there is no behavior written in the base method and just because it was being overridden without any invocation of base methods, it was detected as a smell. In other words, intuitively the authors could conclude even such code smells are primarily being thought about as syntactic where in the tool is just looking for redundant names in the superclass and subclass. Ideally, the tool should check if there is any meaningful behavior attached to the base method and only then should it be detected as a smell.

The dummy example (shown in Listing 3) was not detected as *feature envy* except by *stench blossom* tool. The probable reason for non detection is the declaration of the phone object inside the `getMobilePhoneNumber` method. Logically may not be detected as a code smell but from a semantic perspective `getMobilePhoneNumber` should be part of "`phone`" class. This issue poses a question of correct semantic code smell detection by tools.

The dummy example (shown in Listing 4) has shotgun surgery code smell. The example shows a common mistake that users commit while creating database connection and querying tables. Users tend to create individual connections and command objects in each of the methods as shown in the example. If we have a connection timeout occurs semantically is make sense to create a utility method that takes SQL as

an argument and returns result set "DBUtility.getList" in this method we will open connection and create SQL statements. InCode and iPlasma tools did not detect this code smell.

Although, several tools detect code smells, they do not consider the semantic intent of the code and hence end up with lot of false positives. Reducing false positives is the first step towards increasing the confidence levels of users and proportionately increasing the usage of refactoring tools.

Listing 1: Feature Envy

```
public class Main {
   public void doCommit(){
      a.commit();
      b.commit();
      c.commit();}}
public class A {
   public void commit() {
       //do something
   } }
public class B {
   public void commit() {
      //do something
   } }
public class C {
   public void commit() {
      //do something
   } }
```

Listing 2: Refused Bequest

```
public class Base{
  protected void m1() { }
}
public class Inherited extends Base {
  protected void m1() { //do something }
}
```

Listing 3: Feature Envy

```
public class Phone {
 private final String unformattedNumber;
  public Phone(String unformattedNumber) {
   this.unformattedNumber = unformattedNumber;
  }
  public String getAreaCode() {
   return unformattedNumber.substring(0,3);
  }
  public String getPrefix() {
   return unformattedNumber.substring(3,6);
  }
  public String getNumber() {
   return unformattedNumber.substring(6,10);
  } }
public class Customer {
 public String getMobilePhoneNumber() {
  Phone m_Phone = new Phone("111-123-2345");
   return "(" + m_Phone.getAreaCode() + ") "
  +  m_Phone.getPrefix() + "-"
  + m_Phone.getNumber();
 } }
```

Listing 4: Shotgun Surgery

```
public List<Employee> getEmployeeList() {
 Connection conn = null;
 Statement stmt = null;
 conn = DriverManager.getConnection(DB_URL,
         USER, PASS);
 stmt = conn.createStatement();
 String sql;
 sql = "SELECT * FROM Employees";
 ResultSet rs = stmt.executeQuery(sql);
 return rs.toList<Employee>();
}
public List<Customer> getCustomerList() {
 Connection conn = null;
 Statement stmt = null;
 Class.forName("com.mysql.jdbc.Driver");
 conn = DriverManager.getConnection(DB_URL,
         USER, PASS);
 stmt1 = conn1.createStatement();
 String sql;
 sql = "SELECT * FROM Customers";
 ResultSet rs = stmt1.executeQuery(sql);
 return rs.toList<Customer>(); }
```

VII. Code Smell Detection Pre-Checks

Based on our study, we recommend some pre-checks specific to particular code smells to improve the correctness of detection:

Feature Envy:

- Are referencing methods from multiple classes. Check if moving a method increases coupling and references to the target class.
- Check if mutual work like object dispose, commit of multiple transactions is accomplished in the method. Is it semantically performing a cumulative task.
- Take domain knowledge and system size into account before detecting smell, use semantic/information retrieval techniques to identify domain concepts

Long Class/ God class:

- Ignore utility classes, parsers, compiles, interpreters exception handlers
- Ignore Java beans, Data Models, Log file classes
- Transaction manager classes
- Normalized results with system size

Long Method:

- Check if large switch blocks are written in the methods if multiple code blocks can be extracted with different functionality

Refuse Parent Bequest:

- Ignore Interface methods, they are meant to be overridden.
- Check if base method has any meaning full behaviour attached to it.

VIII. Limitations & Future Work

Our initial study has provided evidence of disagreement towards the detected code smell from tools by our human subjects. We presented sample code for incorrectly detected code smells and semantic smells that were not detected by the tools. The authors strongly felt the need of taking semantic intent of the code into consideration while detecting smells and proposing (semi) automatic refactoring.

The disagreement in detected code smells correlates to the confidence levels of users. We saw that the results from all the users were not the same for the same code smell detection tools. So, the accuracy of these results can always be questioned. The results of Fontana et al. were used for cross-validation of our work. For the degree of disagreement to the detected code smells, we took an average of the overall disagreement report by the users. But, code smell disagreement is subjective until a standardized method for detection and measurement is proposed.

As a future work we would to extend our experiment with more industry level users. We would like to share the disagreement detected by our human subjects with the actual developers of the system to validate our findings. Towards understanding the correctness of the semantic code smell we would like to compare detection logic by tools.

REFERENCES

[1] Code smell evaluation template. http://bit.ly/1WBfZPy. [Online; accessed 30-June-2015].

[2] Code smells detected by human subjects & their disagreements. http://bit.ly/1BNJ6KS.

[3] Detailed profile of our human subjects. http://bit.ly/1KiuEvY. [Online; accessed 30-June-2015].

[4] D. Campbell and M. Miller. Designing refactoring tools for developers. In *Proceedings of the 2Nd Workshop on Refactoring Tools*, WRT '08, pages 9:1–9:2, New York, NY, USA, 2008. ACM.

[5] V. Ferme, A. Marino, and F. A. Fontana. Is it a real code smell to be removed or not? In *International Workshop on Refactoring & Testing (RefTest), co-located event with XP 2013 Conference*, 2013.

[6] F. Fontana, E. Mariani, A. Morniroli, R. Sormani, and A. Tonello. An experience report on using code smells detection tools. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 450–457, 2011.

[7] M. Fowler. *Refactoring: improving the design of existing code*. Pearson Education India, 1999.

[8] J. Mahmood and Y. Reddy. Automated refactorings in java using intellij idea to extract and propogate constants. In *Advance Computing Conference (IACC), 2014 IEEE International*, pages 1406–1414, 2014.

[9] M. MÃntylÃ and C. Lassenius. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering*, 11(3):395–431, 2006.

[10] S. Olbrich, D. Cruzes, and D. I. Sjoberg. Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10, 2010.

[11] W. F. Opdyke and R. E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Symposium on Object-Oriented Programming Emphasizing Practical Applications, September 1990.*, 1990.

[12] A. Ouni, M. Kessentini, H. Sahraoui, and M. Hamdi. Search-based refactoring: Towards semantics preservation. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 347–356, Sept 2012.

[13] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia. Do they really smell bad? a study on developers' perception of bad code smells. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 101–110, Sept 2014.

[14] G. H. Pinto and F. Kamei. What programmers say about refactoring tools?: An empirical investigation of stack overflow. In *Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools*, WRT '13, pages 33–36, New York, NY, USA, 2013. ACM.

[15] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. The qualitas corpus: A curated collection of java code for empirical studies. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 336–345, 2010.

[16] W. C. Wake. *Refactoring Workbook*. Addison-Wesley Longman, Publishing Co., Inc., Boston, MA, USA, 1 edition edition, 2003.